Parallel In-Memory Trajectory-based Spatiotemporal Topological Join

Suprio Ray*, Angela Demke Brown*, Nick Koudas*, Rolando Blanco† and Anil K. Goel†
*Department of Computer Science, University of Toronto. Email: {suprio, demke, koudas}@cs.toronto.edu
†SAP Canada, Waterloo. Email: {rolando.blanco, anil.goel}@sap.com

***Abstract*—The explosive growth of spatiotemporal and spatial data is fueling the emergence and growth of many spatiotemporal applications. Many of these applications are characterized by complex spatiotemporal queries. An important category of such queries is the trajectory-based spatiotemporal topological join queries, which combine a trajectory dataset and a spatial objects dataset based on spatiotemporal predicates. Although these queries have many important use cases, they have not received much attention from the research community.**

We systematically evaluate several applicable in-memory spatiotemporal topological join algorithms, using existing trajectory index (TB-tree) and spatial index (STR). We show that even the best among these algorithms is long running and not scalable. To address the performance problems of these algorithms we introduce PISTON, a parallel in-memory indexing system targeted for spatiotemporal topological join. With extensive evaluations, we demonstrate that even the single-threaded performance of PISTON is significantly better than the applicable approaches that use existing trajectory and spatial indexes. Moreover, the parallel performance of PISTON is orders of magnitude better than these approaches.

I. Introduction

The proliferation of sensors, RFID tags, GPS-equipped mobile devices and smart metering is driving the rapid rise in volume of spatiotemporal data. This is fueling many emerging applications such as location-based games and social networks, location aware search and personalized advertising and weather services. So the efficient management, storage and retrieval of time-stamped location data have become ever more important.

The trajectory-based queries are important classes of spatiotemporal queries [20]. Trajectory-based topological queries deal with the whole or part of the trajectory of a moving object. Given a set of trajectories *R* and another set of spatial objects (polylines or polygons) *S*, a trajectory-based topological join query combines these two data sets on spatiotemporal predicates *P*, such as *enters*, *crosses* or *leaves*. An example of such a query is to select all trajectories that cross a set of polygons during a given time interval in the past. These queries are important in many Location-Based Services (LBS) applications. A food and beverage products company, such as Pepsi or Coca-Cola that delivers drinks from manufacturing plants to stores, may like to create polygons around the plants and stores. In LBS these polygons are often known as *geofences*. The company may want to know the details about the trajectories of their delivery trucks that enter and leave the geofences during rush hours. Such information could then be used to optimize truck routes and in turn save on fuel costs and reduce stop times. Similar use-cases can be found with various other industries including utility services and transportation. Some of these fleets can have tens of thousands of vehicles [7]. New use-cases are also emerging. For instance, RFID tags with GPS support can be used to track perishable goods, such as seafood, and the determination and optimization of routes can reduce wastage. Similarly, GPS-equipped mobile phones can be used track the trajectories of people in urban areas and offer personalized services.

It has been shown [22] that spatial join queries are compute intensive, leading to long query latencies. Although the study of spatial joins have received a lot of attention [17], [18], [22], not much work has has yet been done with the spatiotemporal topological join queries. To evaluate the performance of spatiotemporal topological join queries, given a dataset of moving object trajectories and polygon objects, we describe different **applicable** in-memory join algorithms. In this process we systematically add in-memory versions of a trajectory index, TB-tree (Trajectory-Bundle tree) [20], and a spatial index, STR [13], on the datasets. We evaluate these algorithms with a moderately large sized trajectory dataset having 100 million location records (corresponding roughly to one month's data for 1000 mobile objects) and a polygon dataset that has 5651 polygons. Even the best among these algorithms, called *Indexed Nested Loop Join with 2 Index* or *INLJ2I*, takes **over 8 hours** when the query time interval is 1000. Our study of these algorithms suggests that not unlike the spatial counterparts, spatiotemporal topological join queries are also compute intensive and long running. Obviously, the query latencies may not be acceptable for real-time use-cases, even with a moderately large data set. Our goal is to support ad hoc historical spatiotemporal topological join queries with real-time response times. We also support joining trajectories with evolving polygons, i.e., polygons that change their size and positions over time.

To improve the performance of spatiotemporal topological join queries we introduce an optimization to INLJ2I. Inspired by the Filter step in spatial join, we introduce a step that uses a trajectory MBR (minimum bounding rectangle) filter before performing the indexed nested loop join. We call this algorithm *Indexed Nested Loop Join (2 Index) with Trajectory Filtering* or *INLJ2I-TF*. INLJ2I-TF performs significantly better than INLJ2I, but it still takes a few hundred to a few thousand seconds to execute the query depending on the query time interval. Both INLJ2I and INLJ2I-TF use in-memory versions of existing indexes, TB-tree and STR. A key issue with the existing approaches to indexing trajectory datasets is limited scalability. We were not able to use TB-tree with a larger trajectory dataset (one with 1 billion location records), as it takes too long to create the index with TB-tree. Furthermore,

existing in-memory spatial indexing techniques are not fast enough with point-in-polygon tests, which is an important step in the processing of spatiotemporal predicates.

To address the performance and scalability issues with these indexes, we propose PISTON (Parallel In-memory trajectory-based Spatiotemporal TOpological joiN), a parallel main memory query execution infrastructure designed specifically to address spatiotemporal join. PISTON does not use any existing trajectory index or spatial index. Rather, we introduce a novel parallel in-memory trajectory index and a parallel in-memory spatial index, that are part of PISTON. The trajectory index offers significantly better performance than existing approaches in supporting high update rates and fast data loading times. The spatial index is optimized for point-in-polygon operations and is very fast for common cases. We experimentally evaluate PISTON against INLJ2I-TF in a single-threaded setting, because TB-tree and STR are not parallel indexes. We show that single-threaded PISTON is two to three orders of magnitude faster than INLJ2I-TF with the trajectory dataset having 100 million records. Further, PISTON is significantly more memory efficient than TB-tree & STR for medium and large datasets. We also show the parallel scalability of multi-threaded PISTON with a much larger trajectory dataset consisting of 1 billion records. Moreover, we evaluate PISTON with three polygon datasets consisting of evolving polygons and show that its performance does not diminish.

Our contributions are as follows:

1) We systematically evaluate applicable main memory algorithms with existing trajectory and spatial indexes and demonstrate that they can be long running and are not suitable for real-time applications.
2) We introduce a parallel in-memory spatiotemporal indexing system (PISTON) that includes a novel in-memory trajectory index and a spatial index. We show that even the single-threaded performance of PISTON is significantly better than the in-memory spatiotemporal topological join algorithms that use existing trajectory and spatial indexes.

The rest of the paper is organized as follows. We discuss spatiotemporal join queries, including predicate evaluation and applicable algorithms, in Section II, We introduce our system in Section III and present evaluation in Section IV. Related works are discussed in Section V and conclusions in Section VI.

II. Spatiotemporal Join Queries

The spatiotemporal queries were classified into two main categories by Pfoser et al. [20]: coordinate-based and trajectory-based. The former type of queries deal with the selection of all objects with respect to a given range (e.g., range queries and nearest neighbor queries). The trajectory-based queries, on the other hand, deal with properties of the trajectory of each individual object, such as topology and direction. The trajectory-based queries were further classified into topological queries and navigational queries.

Spatial join queries are used to coalesce two different datasets based on a spatial predicate. In the case of trajectory-based spatiotemporal topological join, one of the two datasets is a trajectory dataset and the other is a dataset of spatial objects such as polylines or polygons. The objects in the spatial dataset could be stationary or evolve over time. Moreover, the predicate must be a spatiotemporal predicate. We formally define the spatiotemporal join operations.

TABLE I: Queries with different predicates and abbreviations

Description (tables involved)	Abbreviations
Trajectory crosses Polygon	Crosses
Trajectory enters Polygon	Enters
Trajectory leaves Polygon	Leaves

A. Trajectory-based Spatiotemporal Topological Join

Definition: Given a trajectory dataset *R*, and spatial objects dataset *S*, a time interval *T* and spatiotemporal predicate *P*, the query operation finds for each $r \in R$, all $s \in S$ such that each *r* and *s* satisfies the predicate *P* within interval *T*. Formally,

$$R \bowtie_{P,T} S \equiv \{(r, s, \{t\}) \mid r \epsilon R \ \wedge \ s \epsilon S \ \wedge \ \exists_{t \subseteq T} P(r(t), s(t))\}$$

Here, *t* is a sub-interval of *T*. Each trajectory *r* is a sequence of triples:

$$r = \{(x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n)\},$$

where $(x_i,\ y_i)$ location coordinates, t_i timestamps and $t_1 < t_2 < ... < t_n$. *S* can be polygon or polyline spatial objects. For the rest of the paper we mean polygons by spatial objects.

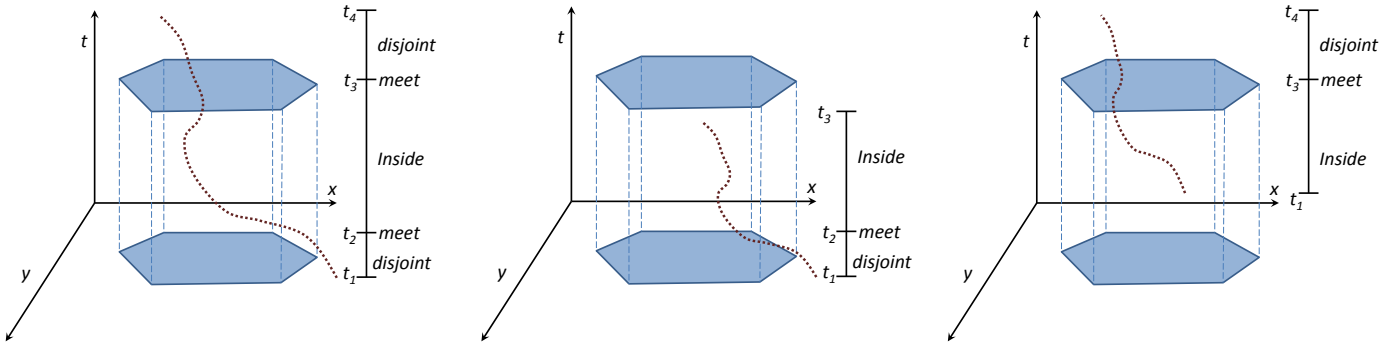
B. Spatiotemporal Topological Predicates

The spatial predicates describe how two spatial objects relate to each other in terms of topological constraints. Several formal models have been been proposed, to characterize these topological relations. The Open Geospatial Consortium (OGC) has adopted one of these, called the Dimensionally Extended Nine-Intersection Model (DE-9IM) [5].

When it comes to spatiotemporal predicates, OGC has still not adopted a standard model. In fact the definition of data models and predicates in this context is a fertile ground of research. Erwig and Schneider [6] proposed a number of spatiotemporal predicates. A number of data models have also been proposed [19]. For our discussion we use three spatiotemporal predicates, which seems to be commonly present in all proposed models. Table I shows the predicates and the queries.

C. Predicates Evaluation

The efficient evaluation of the predicates is necessary to be able to use them in spatiotemporal join queries in a real database system. There have been a few research projects, such as the work by Forlizzi et al. [8], that dealt with the representation of moving objects. The main idea behind these works is to describe the moving objects using *sliced representation* or *unit representation*. In this approach, the temporal evolution of an object is decomposed into a sequence of *slices* or *time units* such that within each unit the evolution is specified by a "simple" function. Schneider [23] proposed a generic mechanism in which the evaluation of a spatiotemporal predicate is described by a sequence of topological relationships that may hold at different time units. For each pair of matching time



(a) Crosses (b) Enters (c) Leaves

Fig. 1: Evaluation of the spatiotemporal predicates

unit a basic topological relation, called the *unit development* is evaluated. For a trajectory of moving points and a polygon object the *unit development* is the outcome of the predicate *point inside polygon*. The concatenated sequence of all the unit developments collected over the entire query interval can then be inspected for certain patterns that evaluate the spatiotemporal topological predicate to be true. This process is illustrated in Figure 1(a), which checks if the spatiotemporal predicate Crosses is valid between a trajectory and a polygon object. The basic topological relations are evaluated at each time unit. Prior to t_2 the trajectory and the polygon are disjoint. The trajectory is inside the polygon between t_2 and t_3, after which they are disjoint again. The predicate evaluation process is shown in Figure 1(b) and Figure 1(c) for Enters and Leaves respectively. For the purpose of evaluating the spatiotemporal topological predicates Crosses, Enters and Leaves between a trajectory and a polygon, it is sufficient to evaluate basic topological relation *point inside polygon* (point-in-polygon test) at each time unit.

D. Applicable Spatiotemporal Join Algorithms

We discuss applicable spatiotemporal topological join algorithms in this section. They are all based on the same principle of spatiotemporal predicate evaluation used by previous works (as outlined in Section II-C). For this discussion we assume that in-memory table *R* is used to store the location updates corresponding to each moving object's trajectory and table *S* is used to maintain the spatial objects. The spatiotemporal predicate is *P*, where the predicates can be Crosses, Enters and Leaves. The query time interval is *T*. We assume that the records in table *R* are similar to "location" records, each with attributes such as timestamp, latitude, longitude, velocity, direction and the moving object id. The records in table *S* have the attributes: *the spatial object geometry, name, start-timestamp, end-timestamp* and *object id*. The start and end timestamps of the spatial objects are used to specify their active lifespan.

The first algorithm, *Nested Loop Join* or *NLJ*, uses no index on either table and so this is a naive approach. The second algorithm uses a trajectory index, I_R, to index the location records in table *R*. There is no index on table *S*. We call it *Indexed Nested Loop Join (1 Index)* or *INLJ1I*. The next algorithm introduces an in-memory spatial index, I_S, on the geometry attribute of *S*. It also uses an in-memory trajectory index, I_R, as before. Since this algorithm uses two indexes we call it *Indexed Nested Loop Join (2 Index)* or *INLJ2I*. We skip the details of *NLJ*, *INLJ1I* and *INLJ2I* due to space constraints.

III. Our approach

First we introduce an optimization into the algorithm INLJ2I. We also examine the performance of applicable spatial join approaches against our optimization. Then we describe our system that we call PISTON (Parallel In-memory trajectory-based Spatiotemporal TOpological joiN).

A. Trajectory Filtering

In algorithm INLJ2I the point-in-polygon test is performed on each coordinate of a trajectory with each spatial object returned by I_S. Inspired by the Filter step of the two-step processing of spatial predicates, we introduce a "trajectory filter" step in the INLJ2I algorithm. This involves checking the minimum bounding rectangle (MBR) of a trajectory for intersection against the spatial index I_S. If this trajectory MBR does not intersect the MBR of any spatial object, then no further action is needed for this particular trajectory. Otherwise the point-in-polygon test needs to performed. We call this algorithm *Indexed Nested Loop Join (2 Index) with Trajectory Filtering* or *INLJ2I-TF*. The algorithm builds the trajectory MBR for each trajectory returned by I_R. Then MBR intersection test is performed to filter out trajectories from further processing. Otherwise, the same processing steps are performed as in algorithm INLJ2I.

B. Initial Evaluation and analysis

To evaluate the spatiotemporal join algorithms that we have described so far, we select two trajectory datasets: Dataset-10mi consists of the trajectories of 10 thousand moving objects (having 10 million location records) and Dataset-100mi has 100 million records for 100 thousand moving objects. Furthers details about them can be found in Table II. We select a spatial dataset: the Arealm polygons from TIGER Texas dataset. The details of this dataset are shown in Table IV. To index the trajectories we chose TB-tree. The polygons were indexed using an efficient spatial index, STR. In both cases we use **in-memory** implementations of the indexes. We use GEOS [9] as the geometry library. We ran the applicable algorithms presented in Section II-D with the Crosses query by varying

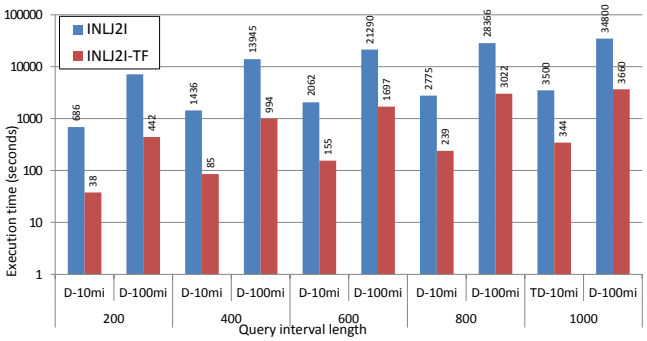


Fig. 2: Execution time of Crosses query with different spatiotemporal join algorithms

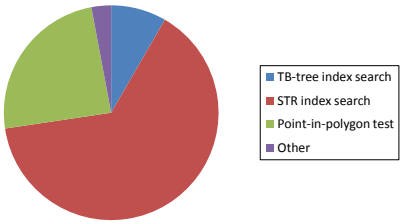


Fig. 3: Breakdown of time with INLJ2I-TF (TB-tree & STR)

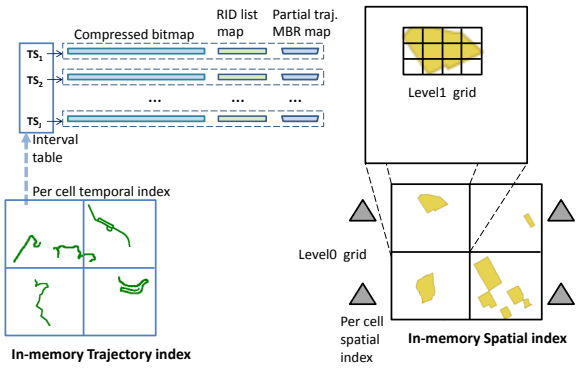


Fig. 4: PISTON overview

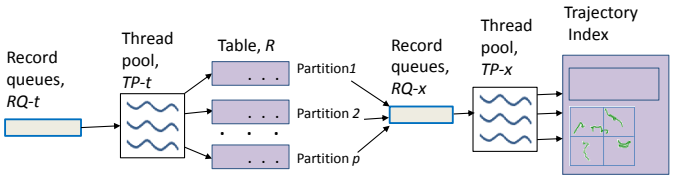


Fig. 5: Trajectory index update process

the query intervals among 200, 400, 600, 800 and 1000. Algorithms NLJ and INLJ1I took too long to be practically used in any application. So we only present the execution times (in seconds) for INLJ2I. We compare its performance against INLJ2I-TF in Figure 2 for both trajectory datasets (label prefix "Dataset" is shortened as "D"). As can be seen, INLJ2I-TF took much less time than INLJ2I in all cases. However, even with the filtering algorithm *INLJ2I-TF took hundreds to thousands of seconds*.

We profiled the execution of INLJ2I-TF with Crosses query for Dataset-10mi and interval 100 using the profiling tool Valgrind ("–tool callgrind" option). Figure 3 shows the breakdown of time reported by Valgrind. As shown, the most time (over 64%) is spent in the spatial index (STR) search. About 24% of time is spent in point-in-polygon test and about 8% in trajectory index search. This suggests that reducing the costs of spatial index and point-in-polygon test are important to improve overall performance.

C. Overview of PISTON

As we have demonstrated, existing approaches to indexing trajectories and spatial objects do not scale. These approaches, such as TB-tree and STR, are usually tree-based and were designed for external memory (disk) join algorithms. Moreover, the spatial indexes do not optimize the basic topological relation such as point-in-polygon test. To improve the performance of trajectory-based spatiotemporal join algorithms, we propose a new parallel, in-memory approach that we call PISTON. The architecture of PISTON is illustrated in Figure 4. PISTON is a grid-based approach that organizes the spatial domain into regular cells. There are two in-memory indexes: the first to index the moving object trajectories and the second to index the spatial objects (e.g., polygons). PISTON's spatial index is specifically optimized for point-in-polygon test. PISTON uses these indexes to implement a parallel in-memory spatiotemporal topological join algorithm, described in Section III-F. The trajectory index is a high performance index designed to handle a high rate of location data updates. The index can handle both a coordinate-based range query and a trajectory-based query. More details about this index are in Section III-D. The spatial index has a two level grid organization: *Level0* and *Level1*. The *Level0* grid of the spatial index uses the same grid organization and is aligned exactly with that of the trajectory index. The purpose of *Level1* grid is to impose grids inside the bounding box of each spatial object. Further details of the spatial index are provided in Section III-E.

D. In-memory Trajectory Index

When a new location update is received, a record is inserted into in-memory table, R (Figure 5), and a unique record id RID is obtained. The table schema: $\{ObjectId, Latitude, Longitude, Direction, Speed, Datestamp\}$. Next the trajectory index is updated. Our in-memory trajectory index discretizes the temporal dimension by maintaining location update information for a fixed length interval i. It discretizes the spatial dimension by imposing regular grid $SGrid_c$ on the spatial domain. Here c=1 to C, where C is the total number of cells. A "cell trajectory", T_c, is part of a moving object's trajectory that is completely within a given grid cell c. Essentially, all the location coordinates corresponding to a moving object's trajectory that are inside a cell boundary belong to T_c. A "partial trajectory", T_i, is comprised of those location coordinates from T_c, whose timestamps are within an interval i. The minimum bounding rectangle (MBR), PM_i, corresponding to a partial trajectory T_i is its "partial trajectory MBR". As illustrated in Figure 6(b), the these MBRs are used to filter out those segments of the trajectory that do not intersect any polygons.

We describe the different components of the in-memory trajectory index, as shown in Figure 4. The main idea behind this is to maintain a per cell temporal index structure for all

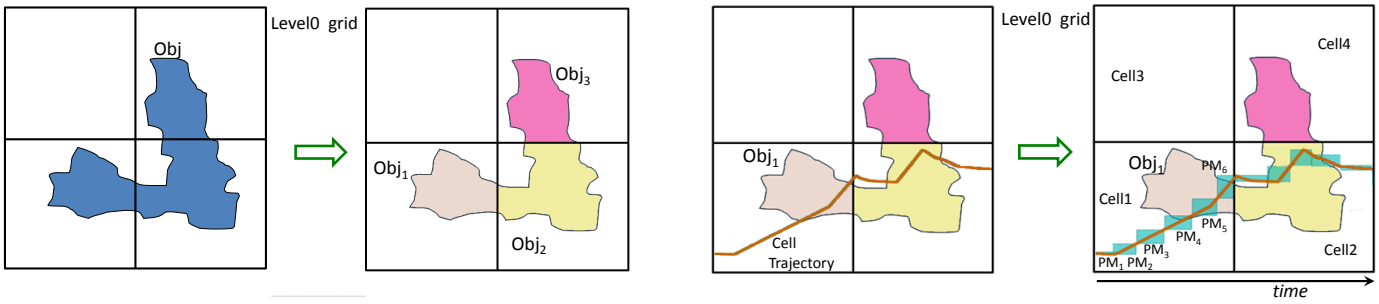


Fig. 6: (a) Splitting objects along tile boundaries and (b) Partial trajectory MBR filtering

the moving objects that visited a grid cell $SGrid_c$. For each temporal interval $i \in I$ during the past N (configurable) days, an entry is maintained in an interval lookup table $Itab_c$ within $SGrid_c$. Each entry in $Itab_c$ corresponds to three data structures: a compressed bitmap $CBmap_{c,i}$, a hashmap $RIDList_{c,i}$ and a second hashmap $PtMBR_{c,i}$. $CBmap_{c,i}$ identifies the the moving objects that were in the grid cell at the given time interval. Therefore, if object m was present during i inside $SGrid_c$, bit position m of $CBmap_{c,i}$ is set. Here m=1 to M, M being total number of objects. We use an insert-optimized compressed bitmap, the details of which are beyond the scope of the paper. Given an RID the actual record storing datestamp, latitude and longitude can be retrieved from table R. The hashmap $RIDList_{c,i}$ is used to maintain for each moving object a list of RIDs corresponding to the location updates sent while at the grid cell c during the time interval i. Essentially the RID list constitutes the partial trajectory for a moving object. To keep the minimum bounding rectangle (MBR) of each partial trajectory, the hashmap *PtMBR* is used, which is keyed by the object id.

Next we describe the trajectory index update workflow. As a new location update is received from a moving object the trajectory index is updated to modify its trajectory information. To support high update rates with thousands of mobile objects, the system is organized to be highly parallel. As shown in Figure 5, an incoming location update is first placed in one of a set of queues *RQ-t* from which it is picked up by a thread in the threadpool *TP-t*. The location table R is partitioned by dividing the RID domain into a number of sub-ranges and a separate *TP-t* thread handles insert into a different partition. Upon inserting a record into the corresponding table partition, it is then enqueued randomly in one among a set of queues *RQ-x*. Since the distribution of the objects and their trajectories could be highly skewed, it is necessary to load-balance the trajectory index update process. The load-balancing scheme involves the threads in the threadpool *TP-x* iterating over the queues *RQ-x*. Each *TP-x* thread inspects the currently chosen queue and processes the next available record. Let the record fields be coordinate *(x,y)*, datestamp ds_t, object id m, and record id *RID*, among others. The target grid cell $SGrid_c$ is determined from its coordinate *(x,y)*. The datestamp ds_t determines the corresponding interval i in the interval table $Itab_c$, where ds_t is the datestamp at time t. If ds_t does not map to an existing interval i in $Itab_c$, a new interval i and related data structures are instantiated. The data structures $CBmap_{c,i}$ and $RIDList_{c,i}$ are updated from the record fields as described earlier. To update $PtMBR_{c,i}$, the old trajectory

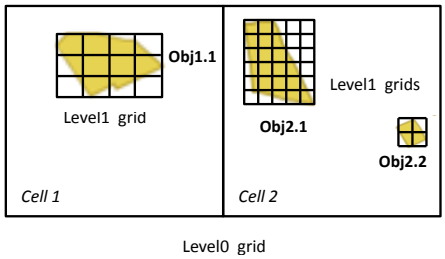


Fig. 7: Different Level1 grid resolutions

MBR of object m, $MBR_{c,i,m}$ is retrieved and is expanded to include new location *(x,y)*.

E. In-memory Spatial Index

We introduce a novel in-memory index for spatial objects that is particularly suitable for point-in-polygon queries. With a tree-based spatial index such as STR, evaluating point-in-polygon query for a given point involves first using the index to retrieve all candidate spatial objects whose MBRs are intersected by the point. Then each candidate spatial object's actual geometry is tested for point containment. However, with our index there is a single processing step that returns all the spatial objects that contain a given point.

Our index is a hierarchical grid based approach. The base level (*Level0*) grid is aligned exactly with the grid of the trajectory index. A higher level (*Level1*) grid is imposed on each of the spatial objects indexed. The resolution of the grid cells at *Level1* is different at different *Level0* cells. This depends on the dimension of the smallest object MBR in a cell, as illustrated in Figure 7. In *Cell2* there are two objects: *Obj2.1* and *Obj2.2*. The smaller of the two is *Obj2.2* and its MBR dimension determines grid cell resolution of *Cell2*. All *Level1* grids imposed on a *Level0* cell have the same resolution. So *Obj2.1* and *Obj2.2* have the same resolution.

A polygon that overlaps multiple *Level0* grid cells requires special treatment. In many of the existing approaches to parallel spatial join [17], [18], such an object is replicated to each cell that it overlaps. We use a technique in which an object is split along the cell boundaries. This is shown in Figure 6(a). In this example the object *Obj* is split into 3 objects: Obj_1, Obj_2 and Obj_3. When a spatial object is split along the cell boundaries of multiple *Level0* cells, different fragments of that object would have *Level1* grid imposed on them with different resolutions. As explained previously, this

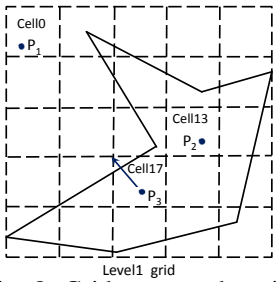


Fig. 8: Grid setup and testing

depends on the dimension of the smallest MBR of that cell. During the processing of the spatiotemporal join query, an aggregation step is used to combine the partial results for the different fragments of an object.

There is a setup process when the spatial objects are indexed for the first time. There are 3 steps in this process. First, each spatial object in table *S* is iterated over and the *Level0* grid cell in which it lies is determined. Objects overlapping multiple cells are also split in this step. In Step 2, each *Level0* grid cell is checked to determine the spatial object with the smallest MBR dimension. Then based on that *Level1* grid is imposed on the spatial objects. Step 3 involves recording the status of the *Level1* cells relative to the spatial objects. With respect to a given polygon, a *Level1* grid cell status could be fully inside, fully outside, or indeterminate. As shown in Figure 8, *Cell0* is fully outside of the polygon and *Cell13* is fully inside. However, *Cell17* is indeterminate with respect to the polygon because one of the edges of the polygon overlaps *Cell17*. The indeterminate cells could also have several polygon edges which overlap the cell. The setup process collects information for each *Level1* grid cell regarding which corner(s) of the cell is inside or outside of the polygon and whether any polygon edge crosses the cell. To deal with overlapping polygons, status information is collected for all polygons and maintained separately within each *Level1* cell. To reduce memory consumption due to the setup process, the cell status is recorded for only those *Level1* cells that are overlapped by the MBR of any spatial object. If no record is found for a *Level1* cell, it means that it is outside of any polygon.

Once the setup process for all the polygons is complete, the spatial index can be used to answer point-in-polygon queries. The griding method [11] suggested a similar point-in-polygon test procedure, that works for a single polygon only. In our approach, this test is performed for the points of the trajectories against all polygons. To check if a point is inside any of the polygons indexed by the spatial index, first the *Level0* cell in which the point lies is determined and then its corresponding *Level1* cell is determined. For a given polygon many of the *Level1* grid cells are either completely inside or outside. Hence a simple look-up is all that is needed to determine if a point is inside the polygon. As a result, this operation could be extremely fast in most cases. For instance, in Figure 8, point P_1 is outside of the polygon because the status of *Cell0* is fully outside. Similarly, point P_2 is inside of the polygon because cell *Cell13* is fully inside.

If the *Level1* cell, inside which the point lies, contains any edges, then a line segment is formed from the test point to

Require: *pt* is a given point and *T* is the query time interval. The procedure returns a list *resultListOfPoly* of all polygons inside which *pt* lies and the polygons are active during *T*.

```
1:  level0Cell ← getLevel0CellForGivenPoint(pt)
2:  level1Cell ← getLevel1CellForGivenPoint(pt,level0Cell)
3:  listPolyRecorded ← getPolygonsRecordedAtCell(level1Cell)
4:  if listPolyRecorded.count = 0 then
5:     resturn NULL
6:  else
7:     for poly in listPolyRecorded do
8:        if level1Cell is fully inside of poly then
9:           if isActiveDuring(poly,T) then
10:             resultListOfPoly.add(poly.id)
11:       else if level1Cell is fully outside of poly then
12:          continue
13:       else
14:          corner ← determine the best cell corner to test
15:          insideFlag ← initFlag(corner)
16:          send a line fragment ray from pt towards corner
17:          edges ← getPolyEdgesRecordedAtCell(poly)
18:          for edge in edges do
19:             if ray intersects edge then
20:                insideFlag ← !insideFlag
21:          if insideFlag and isActiveDuring(poly,T) then
22:             resultListOfPoly.add(poly.id)
23:    return resultListOfPoly
```

Fig. 9: **Procedure evalPntInPoly**

one of the corners of the cell. This is illustrated by an arrow pointing from point P_3 to the top-left corner of cell *Cell17* in Figure 8. This line segment is tested for intersection against all recorded polygon edges for the cell. Since the state of the cell corner (whether inside or outside) is known, the number of times this line crosses the edges of the polygon determines if the point lies inside or outside of the polygon. The steps of this procedure to perform a point-in-polygon test is shown in Figure 9. To support evolving polygons, such that they can change their locations and sizes over a period of time, the spatial index supports multiple versions of the same spatial object. Each version is associated with an active lifespan, with start-timestamp and end-timestamp. The lifespan must be active during query interval *T* for it to be included in the return result.

F. A Parallel In-memory Join Algorithm

PISTON introduces a parallel in-memory spatiotemporal topological join algorithm. Like any other parallel task execution, load-balancing is needed to evenly distribute the query workload to the worker threads. This could be based on static or dynamic work assignment. PISTON follows the dynamic approach with a master/slave model, where the master is called *Manager* and the slaves *Worker*. For each query job the Manager creates many tasks, one for each grid cell, and enqueues them into a synchronized queue. Each Worker thread picks the next task from the queue and performs the spatiotemporal join on the objects from both tables belonging to the corresponding cell.

The sketch of the algorithm is shown in Figure 10. The Manager creates and enqueues a task *STJTask* for each cell (lines 1-3). Then it waits for the Workers to complete. A Worker retrieves the next task from the queue and with the cell id it retrieves the partial trajectory information for that

grid cell from the trajectory index PiI_R (line 19-21). The trajectory index maintains information for all valid active time intervals in memory. For every time interval it is necessary to inspect the "partial trajectory" of each moving object. For each such moving object indexed by the trajectory index, the corresponding "partial trajectory MBR" is retrieved. Then this MBR is checked against the spatial index PiI_S to see if the *MBRIntersects* spatial predicate is satisfied (lines 23-27). This process is similar to the Filter step of the two step Filter and Refinement in regular spatial predicate evaluation. If the object's trajectory MBR does not satisfy the MBRIntersects predicate, no further action is needed for that partial trajectory. Otherwise, every coordinate of that partial trajectory is tested against the spatial index *PiI*$_S$ to do a point-in-polygon test (line 32). The details of Procedure *evalPntInPoly* is in Figure 9. The index returns **all** spatial objects (polygons), $\{s_t\}$, that contain the coordinate. Each such spatial object becomes part of the moving object's *unit development* sequence (lines 32-34). This process is illustrated in Figure 6(b). As can be seen, the "partial trajectory MBRs" PM_1, PM_2 and PM_3 do not intersect the MBR of object *Obj1* and hence can be ignored. Since PM_4, PM_5 and PM_6 satisfies *MBRIntersects* with *Obj1*, only those points in their corresponding "partial trajectories" need to be further processed.

Once all the Workers have processed all the grid cells, the Manager iterates over each task that was scheduled and retrieves partial *unit development* sequences. For each moving object the partial sequences are merged to create the complete *unit development* sequences (line 7-12). Finally, for each such sequence the spatiotemporal predicate *P*, such as Crosses, Enters or Leaves, is evaluated (lines 14-17 in Figure 10).

IV. Experimental Evaluation

In this section we evaluate PISTON. We first describe the experiments involving polygon dataset with shapes that are static and next we conduct experiments with evolving polygons dataset. The trajectory dataset remains the same in both cases.

A. Dataset with static polygons

1) Experimental setup: We use a real-world spatial objects dataset that contains the geographical features of of Texas, drawn from the TIGER® data [25]. Table IV shows the details of this dataset (table *S*). To generate the trajectory dataset (table *R*), the polyline shapefiles of Texas from the TIGER® dataset were fed into the mobility trace generator MOTO [14]. As shown in Table II, we generated the trace files for 3 different sizes. The largest trajectory dataset, Dataset-1bi, contains one billion location records for 1 million moving objects. The different configuration parameters used to generate the trajectory dataset is shown Table III. The experiments were conducted on a machine having 256 GB memory, 8 Intel Xeon processors with 64 cores, each running at 1064 MHz.

2) Comparison with existing trajectory and spatial index: In Section II-D we described algorithms *NLJ*, *INLJ1I* and *INLJ2I*. Then we presented an optimization to INLJ2I, called INLJ2I-TF. They were implemented with trajectory index TB-tree and spatial index STR. We use **INLJ2I-TF results as the baseline to compare against PISTON**, because INLJ2I-TF performs significantly better than *NLJ*, *INLJ1I* and *INLJ2I*.

Require: *R* and *S* are the 2 tables. *PiI*$_R$ is PISTON's trajectory index on table *R*; *PiI*$_S$ is PISTON's spatial index on table *S*. The join predicate is *P*, query time interval is *T* and *mobjDevelMap* is a map containing the unit development sequence of each object.

Manager:

1: **for** *cellId* in *cellList* **do**
2: Create a new *STJTask* with *cellId*
3: *TaskQueue.push(STJTask)*
4: *TaskList.add(STJTask)*
5: {//Wait for all tasks to complete}
6: ...
7: **for** *STJTask* in *TaskList* **do**
8: *partialUnitDevelSeq* ← *STJTask.getPartialUnitDevelSeq()*
9: **for** $\{O_i$,*partialDevelMap*$_i\}$ in *partialUnitDevelSeq* **do**
10: *mObjDevel* ← *mObjDevelMap.get*(O_i)
11: {//Merge to the unit development sequence}
12: *mObjDevel.merge*(*partialDevelMap*$_i$)
13: *mObjList* ← *mObjDevelMap.getKeys()*
14: **for** *mObj*$_i$ in *mObjList* **do**
15: *mObjDevel* ← *mObjDevelMap.get*(*mObj*$_i$)
16: {//Evaluate spatiotemporal predicate: crosses, enters, leaves etc.}
17: *result*$_i$ ← *evalSTPredicate(P,mObjDevel)*

Worker:

18: **while** true **do**
19: *STJTask* ← *TaskQueue.pop()*
20: *cellId* ← *STJTask.cellId()*
21: *partialSTIndex* ← *PiI*$_R$*.getPartialSTIndex(cellId)*
22: {//Do for each interval and each moving object that was in the cell during the interval}
23: **for** $\{interval_j$,*partialTrajInfo*$_j\}$ in *partialSTIndex* **do**
24: **for** $\{O_i$,*objTrajInfo*$_i\}$ in *partialTrajInfo*$_j$ **do**
25: *partialTrajMBR*$_i$ ← *objTrajInfo*$_i$*.getPartialTrajMBR()*
26: *partialTraj*$_i$ ← *objTrajInfo*$_i$*.getPartialTraj()*
27: **if** *PiI*$_S$***.MBRIntersects***(partialTrajMBR*$_i$*)* **then**
28: **for** r_t in *partialTraj*$_i$ **do**
29: l_t ← *getLocation*(r_t)
30: ts_t ← *getTimestamp*(r_t)
31: **if** ts_t is in *T* **then**
32: $\{e_{l,s,t}$,l_t,$\{s_t\}$,*t*$)\}$ ← *PiI*$_S$***.evalPntInPoly***(l_t,*T*)
33: {//Add to partial unit development sequence}
34: *mObjDevel.addPartUnitDevelSeq*($e_{l,s,t}$,l_t,$\{s_t\}$,*t*)

Fig. 10: **Algorithm PISTON**

TABLE IV: Details of polygon dataset

Dataset	Database table	Geometry	Cardinality
Texas	Arealm	polygon	5651

a) Execution time: Since INLJ2I-TF does not support intra-query parallelism we use **single-threaded** execution of PISTON. We execute the Crosses query with Dataset-10mi and Dataset-100mi. We vary the query interval 200, 400, 600, 800 and 1000. The results for Dataset-10mi can be seen in Figure 11(a), and for Dataset-100mi in Figure 11(b). In both cases, single-threaded PISTON is 2 to 3 orders of magnitude faster than INLJ2I-TF. We were not able to evaluate INLJ2I-TF with Dataset-1bi, because TB-tree takes too long to load and create index.

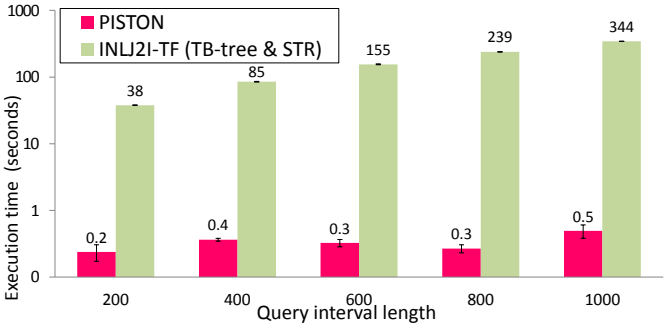
b) Memory usage: Figure 12 shows the memory usage of PISTON against TB-tree & STR. With the smallest dataset Dataset-10mi, PISTON's memory consumption is higher than that TB-tree & STR. But with the moderate sized Dataset-100mi PISTON requires half as much as memory. With the largest dataset Dataset-1bi, TB-tree & STR approach requires

TABLE II: Details of trajectory dataset

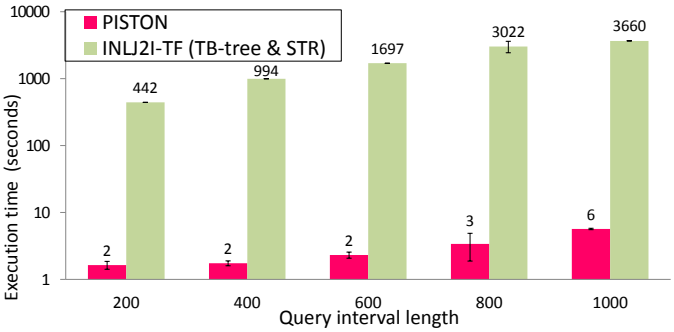
Dataset name	Num. of objects	Num. of records	Size on disk
Dataset-10mi	10,000	10 million	445 MB
Dataset-100mi	100,000	100 million	4.5 GB
Dataset-1bi	1,000,000	1 billion	46 GB

TABLE III: Trajectory data generation

Parameter	Settings
Spatial domain	1251km x 1183km
Num. of polylines	56832846
Duration	1000 *(timestamps)*
Update frequency	10 *seconds*
Updates (num. of records)	10 mi, 100 mi, **1 bi**



(a) Dataset-10mi



(b) Dataset-100mi

Fig. 11: Execution times of query Crosses: PISTON vs NLJ2I-TF (TB-tree & STR)

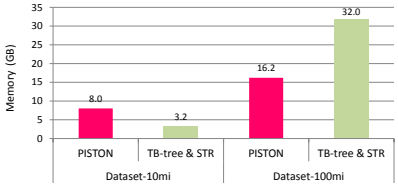


Fig. 12: Memory usage: PISTON vs TB-tree & STR

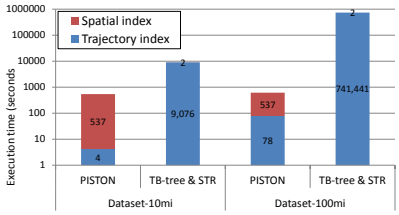


Fig. 13: Index creation time with: PISTON vs TB-tree & STR

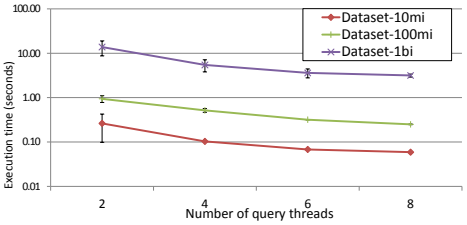


Fig. 14: Execution times of Crosses: vary number of threads (interval length 1000)

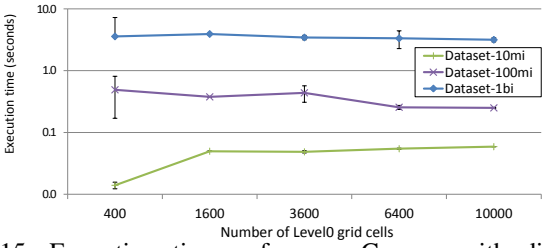


Fig. 15: Execution times of query Crosses with different number of Level0 grid cells (interval length 1000, 8 cores)

more memory than the machine's available RAM capacity. Due to the preprocessing step PISTON'S spatial index requires more memory than the spatial index STR, that stores only object MBRs. On the other hand, due to using compressed bitmap PISTON's trajectory index is much more memory efficient than TB-tree. This suggests that with PISTON's memory usage scales well with Big Data.

c) Index creation time: To compare the overhead associated with PISTON against TB-tree & STR approaches, we report the time to create the trajectory index from the trajectory dataset, as well as, the time to create the spatial index from the polygon dataset. Here the polygon dataset remains the same, but trajectory dataset is changed to Dataset-10mi and Dataset-100mi. With PISTON the spatial index creation time (including the setup steps for polygons) is longer than the trajectory index creation time. With TB-tree & STR approach, the time taken to create the trajectory index TB-tree dominates the overall time. Figure 13, for Dataset-10mi the index creation time is more than an order of magnitude longer with TB-tree & STR compared to PISTON. With Dataset-100mi, TB-tree & STR is more than two orders of magnitude longer than PISTON. In fact, we *could not use TB-tree to generate index for Dataset-1bi*, because it takes very long.

3) Multi-threaded scalability of PISTON: So far PISTON was used in a single-threaded setup. To evaluate the multi-threaded performance of PISTON we executed the Crosses query with 2, 4, 6 and 8 cores. We use all three trajectory datasets: Dataset-10mi, Dataset-100mi and Dataset-1bi. The query interval length was 1000. Figure 14 shows the execution times (in log scale) with different number of query execution threads. As can be seen, with all three datasets PISTON shows near linear speedup with the number of threads from 2 to 6. From 6 threads to 8 threads the reduction in execution time slows down. However, with 8 threads the query execution time is less than a second for both Dataset-10mi and Dataset-100mi; and for Dataset-1bi it is about 3 seconds. Due to the small query execution times, adding more threads offers diminishing returns. Note, that the 8-thread performance of PISTON with Dataset-100mi and interval 1000 represents a **speedup of 14,640X** over INLJ2I-TF, which is quite significant. Note that,

we limit the evaluation to 8 cores even though the machine has more cores. Because, the query execution times with Dataset-10mi and Dataset-100mi are already less than 1 second and using more cores does not improve the performance significantly.

4) Handling skew: The distribution of the trajectories and the spatial objects can be highly skewed. In all previous experiments, we chose PISTON's Level0 grid dimension to be 100x100 (10000 cells). To show how PISTON manages skew, we vary the number of Level0 grid cells to 400, 1600, 3600, 6400 and 10000 (corresponding to dimensions 20x20, 40x40, 60x60, 80x80 and 100x100). We executed the Crosses query with 8 cores for query interval length 1000 using all three trajectory datasets: Dataset-10mi, Dataset-100mi and Dataset-1bi. Figure 15 shows the execution times (in log scale). As can be seen, for the two larger datasets the query performance remains relatively stable with the varying number of grid cells. For the smallest dataset of Dataset-10mi, the query execution time is the lowest when the number of grid cells is 400. This is expected because for this dataset, the max execution time is less than 0.1 second and the extra overhead of processing the cells is the lowest when the number of grid cells is the least. PISTON's trajectory index uses an adaptive load-balancing algorithm similar to in [21]. Its spatial index does not replicate objects, but rather splits objects along tile boundaries. As a result PISTON does a good job of handling the skew.

5) Different spatiotemporal predicates: We executed the queries along with 8 threads for different predicates: Crosses, Enters, Leaves. The interval length was 1000. All three trajectory datasets were used. The results (omitted) show that the performance of the queries was very similar regardless of the predicate for any trajectory dataset.

B. Dataset with evolving polygons

In all previous experiments the polygon objects were fixed. They did not move or change their sizes. There are many scenarios where the objects could evolve over time, such as in urban and agricultural land-use. Other use cases include political boundary changes and geographical changes due to glacial movements, deforestation or water-level changes etc. Next we evaluate PISTON with datasets in which the polygons also evolve with time in terms of changes in location and size. The trajectory datasets remain the same as in Table II.

1) Experimental setup: To generate the moving polygons dataset, we utilized the GSTD tool [24]. While generating polygons dataset with GSTD we used the spatial dimensions of Texas to exactly match with the trajectory dataset's spatial dimensions. The configuration parameters are shown in Table V. GSTD was used to generate evolving polygon datasets for three different distributions: Random, Skewed and Gaussian. Visual representations of the datasets are shown in Figure 18.

2) Different distributions: To compare the execution times of spatiotemporal topological join query for Crosses predicate with the different distributions, we executed the query with three polygon datasets generated by the GSTD tool as described above. We varied the trajectory datasets by using Dataset-10mi, Dataset-100mi and Dataset-1bi. Figure 16 shows the execution times of these queries. In all cases the combination of a given trajectory dataset and the polygon dataset with Gaussian distribution took the longest times. And, it look the shortest times with polygon dataset having Skewed distribution. To understand this result, the trajectory dataset Dataset-10mi is visualized in Figure 18(a). In the case of polygons generated with Skewed distribution (Figure 18(c)), the majority of the polygons are clustered in the top-left corner of the region. When this dataset is joined with the trajectory dataset, most of them are filtered out and very few polygons are actually joined with the trajectories. However, with Gaussian distribution, the majority of the polygons are situated about the center of the region (Figure 18(d)), where most of the trajectories are also clustered. Hence, many of the polygon-trajectory pairs pass the partial trajectory MBR based filtering into the actual predicate evaluation stage. Figure 17 shows the multi-threaded scalability with PISTON for the polygon dataset generated with Gaussian distribution. The trajectory dataset is the Dataset-1bi. PISTON shows near linear speedup as the number of threads are varied from 2 to 6.

V. Related work

Several projects looked into approaches to spatial join. Also a large body of research explored temporal join. But only a few projects investigated spatiotemporal join. We present previous works related to spatiotemporal join, trajectory index and spatial index.

a) Spatiotemporal Join and Trajectory Index: Most of the past research projects in spatiotemporal query and indexing addressed coordinate-based queries. A survey of recent developments in such indexing approaches can be found in [15]. In our previous work we introduced a parallel in-memory index for spatiotemporal range queries [21]. The few projects that looked into spatiotemporal join dealt with problems that are orthogonal to trajectory-based topological join. For instance, Chen and Patel [4] proposed a framework for trajectory distance join and trajectory k Nearest Neighbor join between two trajectory datasets. Iwerks et al. [12] introduced an algorithm to maintain a dynamic view of the "spatial semijoin" results as time progresses. Bakalov et al. [1] addressed the problem of identifying all pairs of similar trajectories between two datasets using symbolic representation.

The work that most closely relates to ours is that of Pfoser et. al [20]. They introduced two spatiotemporal indexes TB-tree and STR-tree, both are based on R-tree. They showed that TB-tree performed better than STR-tree with trajectory-based queries. The main idea behind TB-tree is to bundle segments from the same trajectory into the leaf nodes of the R-tree. In their paper they focussed on range and time-slice queries, but provided no evaluation or result with the topological queries. The Scalable and Efficient Trajectory Index (SETI) [3] is another trajectory index that uses a two level organization to disjoint the spatial from temporal indexing. SETI partitions the 2D space into disjoint hexagon cells which remain static during the structure's lifetime. Like [20], the SETI paper only evaluated range and time-slice queries. Recent works, including [10], [16] addressed the problem of how to optimally split trajectories to improve range query performance. But, none of them addressed trajectory-based topological join.

b) Spatial Index: Since one of the two datasets in a trajectory-based spatiotemporal topological join is a spatial

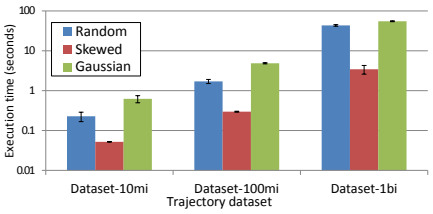


Fig. 16: Execution times of Crosses: different distributions (single threaded)

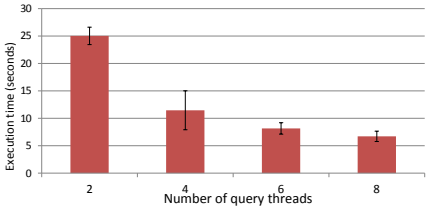
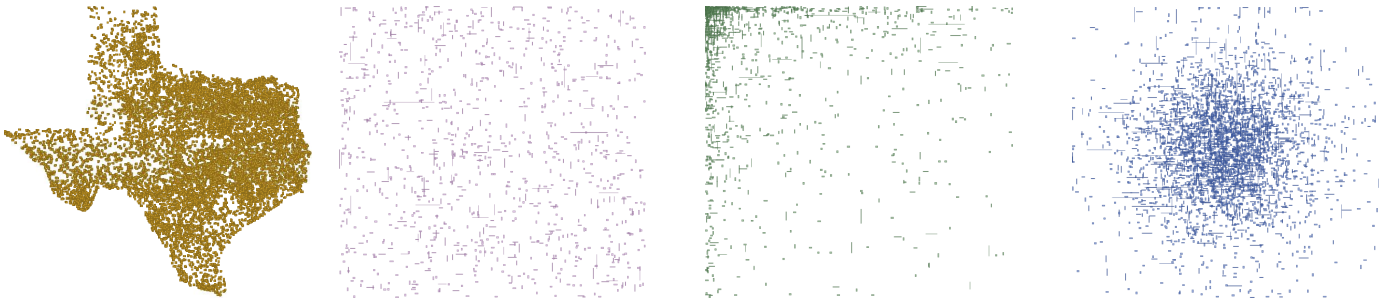


Fig. 17: Execution times of Crosses: Gaussian (Dataset-1bi, multi-threaded)

TABLE V: Evolving polygons dataset generation settings

Parameter	Settings
Space domain	1251x1183 km^2
Num. of objects	1000
Time duration	1000
Distribution	Random, Skewed, Gaussian



(a) Trajectory dataset Dataset-10mi (b) Random distribution (c) Skewed distribution (d) Gaussian distribution

Fig. 18: Distribution of polygon objects

dataset (the other is a trajectory dataset), we survey works related to indexing spatial objects. The most well-known spatial index is the R-tree. It is susceptible to degraded search performance due to node overlap and multi-path probing. A number of heuristics have been proposed to reduce the node overlap. The R*-tree [2] uses a custom split heuristics by removing and reinserting a portion of the entries when a node overflow occurs. The STR (Sort-Tile-Recursive) [13] sorts the records in each axis and splits the space along each dimension into strips with similar number of records.

VI. Conclusions

With the explosive growth of spatiotemporal data and the increasingly popularity of Location-Based Services (LBS), there is growing impetus to deal with more complex spatiotemporal queries. Whereas spatial join received wide attention from the research community, it is not the case with spatiotemporal topological join. We have showed that such queries can be long running even with a moderately large trajectory dataset.

We introduce PISTON, a parallel in-memory spatiotemporal query processing system that consists of a high performance spatial index and a trajectory index. With extensive evaluation, we have demonstrated that our system achieves several orders of magnitude better performance than the applicable algorithms using existing trajectory and spatial index.

References

[1] P. Bakalov, M. Hadjieleftheriou, E. Keogh, and V. J. Tsotras. Efficient trajectory joins using symbolic representations. In *MDM*, pages 86–93, 2005.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.

[4] Y. Chen and J. M. Patel. Design and evaluation of trajectory join algorithms. In *SIGSPATIAL GIS*, pages 266–275, 2009.

[5] E. Clementini and P. Di Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, 90(1-4):121–136, April 1996.

[6] M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):881–901, 2002.

[7] Top 300 commercial fleets. http://www.fleet-central.com/TopFleets/.

[8] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *SIGMOD*, pages 319–330, 2000.

[9] GEOS. http://http://trac.osgeo.org/geos/.

[10] M. Hadjieleftheriou, G. Kollios, J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB Journal*, 15(2):143–164, June 2006.

[11] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, 1994.

[12] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of spatial semijoin queries on moving points. In *VLDB*, pages 828–839, 2004.

[13] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pages 497–506, 1997.

[14] MOTO (Moving Objects Trace generatOr). http://moto.sourceforge.net/.

[15] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE Data(base) Engineering Bulletin*, 33:46–55, 2010.

[16] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Trans. on Knowl. and Data Eng.*, 19(5):663–678, May 2007.

[17] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[18] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *SIGSPATIAL GIS*, pages 54–61, 2000.

[19] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis. Literature review of spatio-temporal database models. *Knowl. Eng. Rev.*, 19(3):235–274, Sept. 2004.

[20] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *VLDB*, pages 395–406, 2000.

[21] S. Ray, R. Blanco, and A. K. Goel. Supporting location-based services in a main-memory database. In *MDM*, pages 3–12, 2014.

[22] S. Ray, B. Simion, A. D. Brown, and R. Johnson. Skew-resistant parallel in-memory spatial join. In *SSDBM*, pages 6:1–6:12, 2014.

[23] M. Schneider. Evaluation of spatio-temporal predicates on moving objects. In *ICDE*, pages 516–517, 2005.

[24] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *SSD*, pages 147–164, 1999.

[25] TIGER®. http://www.census.gov/geo/www/tiger.