IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

GeoFlink: An Efficient and Scalable Spatial Data Stream Management System

SALMAN AHMED SHAIKH[1], HIROYUKI KITAGAWA[1,2] (Member, IEEE), AKIYOSHI MATONO[1], KOMAL MARIAM[3,*] AND KYOUNG-SOOK KIM[1]

[1]Artificial Intelligence Research Center (AIRC), National Institute of Advanced Industrial Science and Technology (AIST) (e-mail: {shaikh.salman, kitagawa.h, a.matono}@aist.go.jp)
[2]International Institute for Integrative Sleep Medicine, University of Tsukuba, Tsukuba, Japan
[3]School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan
[*]This work was done during an internship at AIRC, AIST, Tokyo, Japan

Corresponding author: Salman Ahmed Shaikh (e-mail: shaikh.salman@aist.go.jp).

ABSTRACT

This era is witnessing an exponential growth in spatial data due to the increase in GPS-enabled devices. Spatial data can be of extreme use to commercial businesses, governments and NGOs if processed timely. Spatial data is voluminous and is usually generated as a continuous data stream. For instance, vehicles' GPS data, mobile subscribers location data, etc. To process such data, highly scalable systems are needed. Apache Spark Streaming, Apache Flink, and Apache Samza are among the state-of-the-art scalable stream processing platforms; however, they lack spatial objects, indexes, and queries support. Besides them, other scalable spatial data processing platforms including GeoSpark, Spatial Hadoop, etc. do not support streaming workloads and can only handle static or batch data. To fill this gap, we present GeoFlink, which extends Apache Flink to support spatial objects, indexes and continuous queries over spatial data streams. To support efficient spatial query processing and effective data distribution across distributed cluster nodes, a gird-based index is introduced. GeoFlink supports spatial range, spatial kNN and spatial join queries on Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon spatial objects. Besides, GeoFlink supports data streams in GeoJSON, WKT, and CSV data formats. A detailed experimental study on real and synthetic spatial data streams proves that GeoFlink achieves significantly higher query throughput than the existing state-of-the-art streaming platforms.

INDEX TERMS GeoFlink, Spatial data, GeoSpatial, Stream processing, Spatial Data Management System, Spatial index, Spatial objects

I. INTRODUCTION

With the increase in the use of GPS-enabled devices, spatial data is omnipresent. Many applications require real-time processing of spatial data, for instance, to provide route guidance in disaster evacuation, for patients tracking to prevent the spread of serious diseases, to support smooth voice and data services to mobile subscribers in all areas, for road traffic monitoring and management, etc. Such applications entail real-time processing of millions of tuples per second. Existing spatial data processing frameworks, for instance, PostGIS [1] and QGIS [2] are not scalable to handle such huge data and throughput requirements, while scalable platforms like Apache Spark Streaming [3], Apache Flink [4], Apache Samza [5], etc. do not natively support spatial data processing, i.e., they lack spatial data objects, indexes, and queries support, resulting in increased spatial querying cost. Besides, there exist a few solutions to handle large scale spatial data, for instance Hadoop GIS [6], Spatial Hadoop [7], GeoSpark [8], etc. However, they cannot handle real-time spatial streams. To fill this gap, we present GeoFlink, which extends Apache Flink to support spatial objects, indexes and continuous queries over spatial data streams.

Indexes are indispensable for efficient query processing and pruning. Spatial data indexes can be broadly divided into 2 types: 1) Tree-based, 2) Grid-based. Unlike static data, stream tuples arrive and expire at a high velocity. Thus, we

need an index with low or zero maintenance cost. Tree based indexes are extensively used for static spatial data processing due to their low data retrieval cost. However, the tree based indexes do not perform well in case of high insertions and deletions as these operations may trigger index restructuring which is a heavy operation. Thus, tree-based indexes are not suitable for data streams due to their high data arrival velocity, which results in high index maintenance cost [9]. On the other hand, given a data boundary, grid indexes have fixed structure and do not require maintenance as new data stream tuples arrive or depart. Therefore, to enable real-time processing of spatial data streams, a light weight logical grid index is introduced in this work. GeoFlink assigns grid-cell IDs to the incoming stream tuples based on which the objects are processed, pruned and/or distributed dynamically across the cluster nodes. GeoFlink supports spatial range, spatial kNN and spatial join queries on Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon spatial objects. Besides, GeoFlink supports data streams in GeoJSON, WKT, and CSV data formats. It provides a user-friendly Java/Scala API to register spatial continuous queries (CQs). GeoFlink is an open source project and is available at Github[1].

```
//Defining dataStream boundaries and index
double minX = 115.50, maxX = 117.60;
double minY = 39.60, maxY = 41.10;
int gridSize = 100;

UniformGrid uGrid = new UniformGrid(
gridSize, minX, maxX, minY, maxY);

// Definig ordinary and query streams
String inputFormat = "GeoJSON";
String dateFormat = "yyyy-MM-dd'T'HH:mm:ss";

DataStream<Point> S1 = Deserialization.
PointStream(oStream, inputFormat, dateFormat);

DataStream<Point> S2 = Deserialization.
PointStream(qStream, inputFormat, dateFormat);

// Real-time Query Configuration
int omegaDuration = 1; // time unit: seconds
QueryConfiguration realTimeConf = new
QueryConfiguration(QueryType.RealTime);
realTimeConf.setWindowSize(omegaDuration);

//Continous join query
DataStream<Tuple2<Point, Point>> joinStream =
new PointPointJoinQuery(realTimeConf, uGrid,
uGrid).run(S1, S2, distance);
```

Code 1. A GeoFlink (Java) code for spatial join query

Example 1 (Use case: Patients tracking)**.** A city administration is interested in monitoring the movement of a number of their high-risk patients. Particularly, the administration is interested in knowing and notifying all the residents in real-time, if a patient happens to pass them within certain distance r. Let $S1$ and $S2$ denote the real-time ordinary residents' and patients' location stream, respectively, obtained through their smart-phones. Then, this query includes real-time join of $S1$ and $S2$, such that it outputs all the $p \in S1$ that lie within r distance of any $q \in S2$. Code 1 shows the implementation of this real-time CQ using GeoFlink's spatial join. The details of each statement in the code is discussed in the following sections.

This paper is an extension of our previous work [10]. The main contributions of this extension are summarized below:

- Grid index extension for LineString and Polygon objects.
- Gird-based distance and r-neighbors computation of Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon spatial objects.
- Real-time and window-based spatial range, kNN and join queries for Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon spatial objects.
- Grid-based pruning for efficient spatial queries processing.
- Extensive experimental study on synthetic and real data streams.

Rest of the paper is organized as follows: Section II presents related work. Section III discusses the essential concepts useful in understanding the later discussion. In Section IV, spatial objects and spatial streams are defined. Section V discusses spatial distance and r-neighbors computation. In Section VI, spatial indexing is discussed and GeoFlink's grid index is presented. Section VII presents the supported spatial queries along with their algorithms. In Section VIII, GeoFlink architecture is presented. In Section IX detailed experimental study is presented while Section X concludes our paper and highlights a few future directions.

II. RELATED WORK

Existing spatial data processing frameworks like ESRI ArcGIS [11], PostGIS [1] and QGIS [2] are built on relational DBMS and are therefore not scalable to handle huge data and throughput requirements. Besides, scalable spatial data processing frameworks, for instance, Hadoop GIS [6], Spatial Hadoop [7], GeoSpark [8], Parallel Secondo [12] and GeoMesa [13], cannot handle real-time processing of spatial data streams. One can find a number of extensions of Spark to support spatial data processing. SpatialSpark [14] leverages Apache Spark's broadcast mechanism to provide partitions based on broadcast space. GeoSpark [8] processes spatial data by extending Spark's native Resilient Distributed Dataset (RDD) to create Spatial RDD (SRDD) along with a Spatial Query Processing layer on top of the Spark API to run spatial queries on these SRDDs. For efficient spatial query processing, GeoSpark creates a local spatial index (Grid, R-tree) per RDD partition rather than a single global index. For re-usability, the created index can be cached on main memory and can also be persisted on secondary storage for later use. However, the index once created cannot be updated, and must be recreated to reflect any change in the dataset due to the immutable nature of RDDs. LocationSpark [15],

[1]GeoFlink @ Github https://github.com/aistairc/SpatialFlink

GeoMesa [13] and Spark GIS [16] are a few other spatial data processing frameworks developed on top of Apache Spark. All these frameworks, like the GeoSpark, do not support real-time stream processing as we do. Apache Spark Streaming [3], Apache Flink [4] and similar distributed and horizontally scalable platforms support large-scale, real-time processing of data streams. However, they do not natively support spatial data processing, i.e., they lack spatial data objects, indexes, and queries support, resulting in increased spatial querying cost.

For real-time queries, Apache Spark introduces Spark Streaming that relies on micro-batches to address latency concerns and mimic streaming computations. Latency is inversely proportional to batch size; however, the experimental evaluation in [17] shows that as the batch size is decreased to very small to mimic real-time streams, Apache Spark Streaming is prone to system crashes and exhibits lower throughput and fault tolerance. Furthermore, even with the micro-batching technique, Spark Streaming only approaches near real-time results at best, as data buffering latency still exists, however, miniscule. Indeed F. Zhang et al. [8] mentions that the architecture of their spatial querying framework could demonstrate a higher throughput if implemented on Apache Flink. Other distributed streaming platforms worth considering are Apache Samza [5] and Apache Storm [18]. Performance comparison by Fakrudeen et al. [19] revealed that both the Samza and Storm demonstrate a lower throughput and reliability than Apache Flink [4]. Thus, we extend Apache Flink, a distributed and scalable stream processing engine, to support real-time spatial stream processing.

To support real-time processing of spatial data streams, Zhang et al. [20] extended Apache Storm. They proposed and implemented distributed spatial indexing for the continuous spatial query processing. The tuple in their work consists of two pairs of coordinates, corresponding to new and old coordinates. Thus, if an object moves greater than some threshold δ distance, the tuple coordinates are updated. For query processing in their work, a hybrid index is introduced, consisting of a primary grid index and a secondary index; where the secondary index corresponds to the objects in a grid cell. The secondary index is either tree- or hash-based and physically stores the moving objects' tuples. The secondary index is mainly maintained to improve the query performance and must be updated as the object moves or updated. In contrast, the grid index in GeoFlink is logical, in the sense that it only assigns a grid ID to the incoming streaming tuples or moving objects and based on the grid ID the objects are processed, pruned and distributed across the cluster nodes. We do not physically store the objects in any data structure or memory, hence no update is required as we receive an updated object location. Hence it enables GeoFlink to achieve higher query throughput.

Another related work is on systems for spatio-textual stream processing [21]–[23]. They deal with streaming tuples which contain geolocations and textual contents. Tornado [21], [22] is a distributed system developed on top of Storm for real-time processing of spatio-textual queries and streams. SSTD [23] enables more variety of continuous queries and snapshot queries as well as continuous ones. These approaches employ sophisticated global indexes and cost models to achieve dynamic load balancing of workers in multiple query and stream processing context. In contrast, GeoFlink utilizes maintenance-free logical gird index and key-based (hash-based) data partitioning scheme inherent in Flink for data distribution. As proved in the experiments in Section IX, GeoFlink can achieve load-balance in processing multiple queries and streams with skews and hotspots if the grid size is chosen carefully.

III. ESSENTIAL CONCEPTS

In this section we present Flink programming model and Flink-Kafka pipeline, which are essential in understanding the later discussion.

A. APACHE FLINK

To support continuous query processing over spatial data streams our platform needs to support features like aggregation, join, windowing, state management, etc. The choice boils down to Apache Spark Streaming and Apache Flink with both the architectures providing basic frameworks to implement the above features. However, as the target of this work is streaming data and continuous queries rather than batch data and one-time queries, Apache Flink is a natural fit for it as it is inherently designed for streaming applications. In the following, we present a quick overview of Apache Flink.

1) Data Collections and Program

Apache Flink uses two data collections to represent data in a program: 1) DataSet: A static and bounded collection of tuples, 2) DataStream: A continuous and unbounded collection of tuples. A Flink program consists of 3 building blocks: 1) Source, 2) Transformation(s)/Operator(s) (in the following, transformations and operators are used interchangeably), and 3) Sink. When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformations. Flink comes with a number of basic transformations such as filter, map, reduce, keyby, join, aggregations, window, etc. Each dataflow starts with one or more data sources and ends in one or more data sinks. The dataflows resemble arbitrary directed acyclic graphs (DAGs). Flink provides seamless connectivity with data sources and sinks like Apache Kafka (source/sink), Apache Cassandra (sink), etc. [4]. In the following, we will discuss Apache Flink from *DataStream* perspective which is the focus of this work.

2) Window Transformation

Window is one of the most important transformations and is regarded as the heart of stream processing. Therefore we discuss this transformation separately. Transformations can be classified broadly into blocking and non-blocking.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

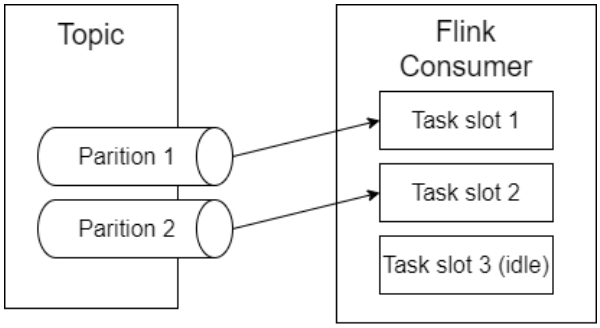


FIGURE 1. The number of kafka topic partitions are less than the number of Flink's task slots. It results in 1 to 1 mapping between the topic partitions and the task slots and causes the remaining task slots to sit idle during distributed query execution. Thus, wasting system resources.

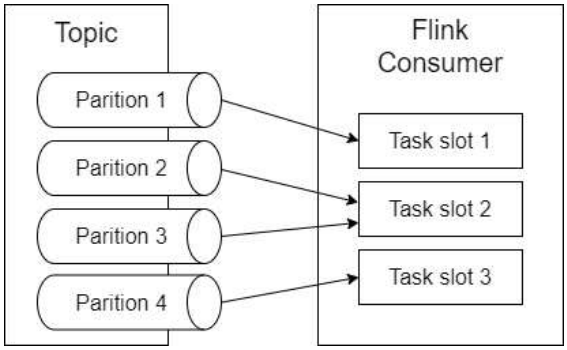


FIGURE 2. The number of kafka topic partitions are more than the number of Flink's task slots. It results in the assignment of multiple topic partitions to a task slot. This causes non-uniform data transfer to some of the task slots. Thus, overburdening a few task slots.

Definition 1 (Blocking operators)**.** Blocking operators require processing of the entire input before an output can be generated. For example, sort, aggregation, join, etc.

Definition 2 (Non-Blocking operators)**.** Non-blocking operators generate output as they receive input tuples without the need to wait for other tuples. For example, filter, map, etc.

Window splits the infinite stream into *buckets* of finite size, over which computations can be applied. Windows are mainly used for blocking operators; however, depending upon applications requirements it can also be used for non-blocking operators. For instance, filter is a non-blocking operator; however, one can implement window-based filter operator to generate periodic output.

Flink supports four types of windows, namely tumbling windows, sliding windows, session windows and global windows. 1) *Tumbling window*: When using this window, a stream is divided into non-overlapping partitions and the stream data is kept only for the current partition. The partition size is specified using a user-defined parameter *window size*. 2) *Sliding window*: In this case, a stream is divided into possibly overlapping partitions of newest events. Similar to the tumbling window, the size of a sliding window is specified using *window size* parameter. An additional *window slide* parameter controls the output frequency. A sliding window is overlapping if *window slide* $<$ *window size*, else non-overlapping. 3) *Session window*: The session window groups tuples by sessions of activity. Session windows do not overlap and do not have a fixed start and end time. Instead a session window closes when it does not receive tuples for a certain period of time, i.e., when a gap of inactivity occurred. 4) *Global window*: A global window assigns all tuples with the same key to the same single global window. Without loss of generality in this work, only tumbling and sliding windows are used.

3) Parallel and Distributed Processing

Programs in Flink are inherently parallel and distributed. During execution, a transformation/operator is divided into one or more subtasks which are independent of one another and execute in different threads that may be on different machines or containers. Flink parallelism depends on the number of available task slots, which are equivalent to the number of CPU cores. In Flink, *keys* are responsible for data distribution across operator instances. All the tuples with the same key are guaranteed to be processed by a single operator instance. In addition, many of the Flink's core data transformations like join, groupby, reduce and windowing require the data to be grouped on keys. Intelligent key assignment ensures the effective (near uniform) data distribution among operator instances and hence leverage the performance offered by parallelism. Many times, data source is a bottleneck in leveraging the performance offered by Apache Flink. Thus, we provide a quick overview of Flink-Kafka pipeline in Section III-B, where Apache Kafka is one of the most common streaming source used with Apache Flink.

4) Workload Fluctuation and Scaling

Streaming jobs usually run for several days or even longer and may experience workload fluctuation during their lifetime. Although some of these fluctuations are more predictable than others, in all cases there is a change in job resource demand that needs to be addressed if we want to ensure the same quality of service. To address the change in job resource demand, Flink supports the following two scaling approaches [24]:

- **Manual:** Manually rescaling a Flink's job has been possible since Flink 1.2 introduced rescalable state, which allows users to stop-and-restore a job with a different parallelism. For example, if our job is running with a parallelism of p=100 and our load increases, we can restart it with p=200 from the savepoint created during shutdown to cope with the additional data.
- **Reactive:** The reactive mode has been introduced in Flink 1.13. In this mode, end-user monitors Flink cluster and add or remove resources depending on some metrics and Flink does the rest, i.e., manages the rescalable states. Reactive mode is a mode where JobManager will

try to use all TaskManager resources available.

Since GeoFlink is built on top of Flink, it supports all the rescaling approaches supported by Flink.

B. FLINK-KAFKA PIPELINE

Apache Flink [4] is mainly used for stream data processing and analytics. In real applications, mostly if not always, the data streams are ingested from Apache Kafka [25], a system that provides durability and pub/sub functionality for data streams. A typical pipeline of Flink and Kafka include data streams being pushed to Kafka, which are then consumed by Flink programs. The outputs of these programs are fed back to Kafka for consumption by other queries, applications or services, written out to distributed file systems, or sent to web frontends [26].

Apache Kafka provides a distributed data streams for efficient processing of Flink jobs. This is achieved by an abstraction in Kafka called *topic*. A topic is a handle to a logical stream of data, consisting of many partitions. Topic partitioning is important in providing distributed data streams to its consumers. Since partitions are assigned to Flink's parallel task instances, when there are more Flink tasks than Kafka partitions, some of the Flink consumers will just sit idle, not reading any data as shown in Figure 1. On the other hand, when there are more Kafka partitions than Flink tasks, Flink consumer instances will subscribe to multiple partitions at the same time as shown in Figure 2. Besides, data streams velocity is unpredictable. One may need to rescale the Flink job, i.e., increase or decrease its parallelism to handle the changing velocity of incoming data stream. Thus, it is very important to partition a topic by keeping in view the maximum available Flink task slots and possible rescaling to avoid wastage of resources. In general, it is recommended to have more partitions than the number of Flink task slots for higher throughput and to avoid task slots' idleness. However, too many partitions can increase latency. Detail discussion on Kafka topic partitioning is beyond the scope of this work. One may refer to the article by Paul Brebner [27] for the detailed discussion on intelligent topic partitioning for higher throughput.

IV. SPATIAL OBJECTS AND SPATIAL STREAMS

This work proposes GeoFlink, which is an efficient and scalable spatial data stream management system. GeoFlink supports the following types of spatial objects in 2D space. Logically, we can deal with multi-dimensional objects in nD space. We focus on 2D space in this paper.

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon

Figure 3 shows the spatial objects listed above. A Point is the simplest spatial object consisting of 2 positional attributes. In geographical coordinate system, the positional attributes are referred as longitude and latitude. A MultiPoint can be defined as a set of Points consisting of multiple Points. A LineString is a connected series of line segments and is also referred as a polygonal chain in geometry, where a line segment is a line bounded by two distinct end points. A MultiLineString consists of multiple disconnected LineStrings. A Polygon is also a connected series of line segments, where the first end point of the first line segment coincides with the second end point of the last line segment, thus, forming a closed shape. A Polygon may also contain one or more holes within it. Like MultiLineString, MultiPolygon consists of multiple disconnected Polygons. In this work, we use ψ to denote a spatial object.

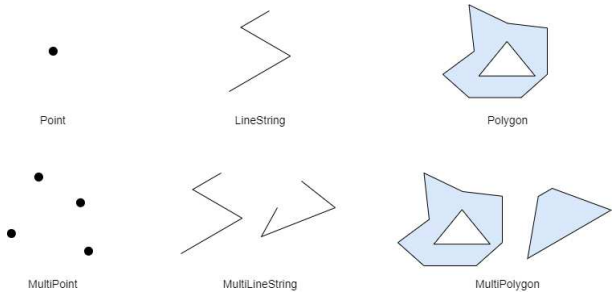


FIGURE 3. GeoFlink supported geometry types

A spatial stream consists of one or more type of spatial objects. Formally, spatial stream can be defined as follows:

Definition 3 (Spatial Stream S)**.** A spatial stream S is a sequence of spatial tuples ordered by their timestamps, where each tuple consists of at-least two attributes, i.e., spatial object (ψ) and timestamp.

To make the discussion simple in this work and without loss of generality, we assume that a spatial stream consists of any one type of spatial object. Thus, a Point stream consists of only Points, a Polygon stream consists of only Polygons, and so on. Furthermore, each stream tuple contains entire spatial object and timestamp, and may contain other attributes, for instance, object identifier, object bounding box, etc.

V. DISTANCE AND NEIGHBORS

In this section we discuss distance computation between different spatial objects and r-neighbors computation, required for the proposed spatial queries discussed in Section VII.

A. SPATIAL DISTANCE

Spatial queries proposed in this work require extensive distance computation between spatial objects. Since GeoFlink supports 6 different types of spatial objects (Figure 3), distance computation between them is required for the execution of the proposed queries. In GeoFlink, Point, LineString and Polygon are treated as special cases of MultiPoint, MultiLineString, and MultiPolygon, consisting of one Point,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

LineString or Polygon, respectively. Namely, we require the following distance computations.

- Point to Point
- Point to LineString
- Point to Polygon
- LineString to LineString
- LineString to Polygon
- Polygon to Polygon

Distance computation between geometrical shapes, also known as similarity measure, is a well-studied topic. There exist a number of distance functions, for instance, Hausdorff metric [28] for Point sets and Polygons, Frechet distance [29] for LineStrings, etc., and their computation techniques in the literature [30] [31]; however, their detailed discussion is outside the scope of this work. GeoFlink uses distance functions provided by JTS Topology Suite [32]. The JTS Topology Suite is a Java API for modeling and manipulating 2-dimensional linear geometry. JTS uses the implicit coordinate system of the input data. The only assumption it makes is that the coordinate system is infinite, planar and Euclidean (i.e. rectilinear and obeying the standard Euclidean distance metric). In the same way JTS does not specify any particular units for coordinates and geometries. Instead, the units are implicitly defined by the input data provided. JTS provides numerous geometric predicates and functions. In this work Euclidean distance is used for distance computation between spatial objects. However, users can write their own distance functions by altering the methods in the GeoFlink's *DistanceFunctions.java* class.

B. R-NEIGHBORS

All of the GeoFlink's spatial queries require neighborhood computation. Thus, given a spatial object ψ and distance r, we define r-neighbor$_{region}(g)$ and r-neighbors(ψ).

Definition 4 (r-distance(μ))**.** For a coordinate μ and distance r, r-distance(μ) = {coordinate ν | distance(ν, μ) <= r}.

Definition 5 (r-neighbor$_{region}(\psi)$)**.** For a spatial object ψ and distance r, r-neighbor$_{region}(\psi)$ = {coordinate ν | ν is contained in some r-distance(μ) where coordinate μ is within (including boundary) object ψ}.

Figure 4 shows r-neighbor$_{region}(\psi)$ of a Polygon query object. Based on r-neighbor$_{region}(\psi)$, we can now define r-neighbors(ψ) as follows.

Definition 6 (r-neighbors(ψ))**.** For a spatial object ψ and distance r, r-neighbors(ψ) = {spatial object γ | r-neighbor$_{region}(\psi) \cap \gamma$ (namely, the spatial intersection area) $\neq \emptyset$ }

Thus, any object which lies within or overlaps the r distance of ψ is r-neighbors(ψ). In Figure 4, objects $o2$, $o5$, $o6$, $o8$, and $o11$ are r-neighbors(ψ). In the following, we discuss r-neighbors(ψ) computation for Point, LineString and Polygon spatial objects. The r-neighbors(ψ) for MultiPoint, MultiLineString, and MultiPolygon can be computed by calculating r-neighbors(ψ) for each spatial object in respective multi-geometry and taking their union.

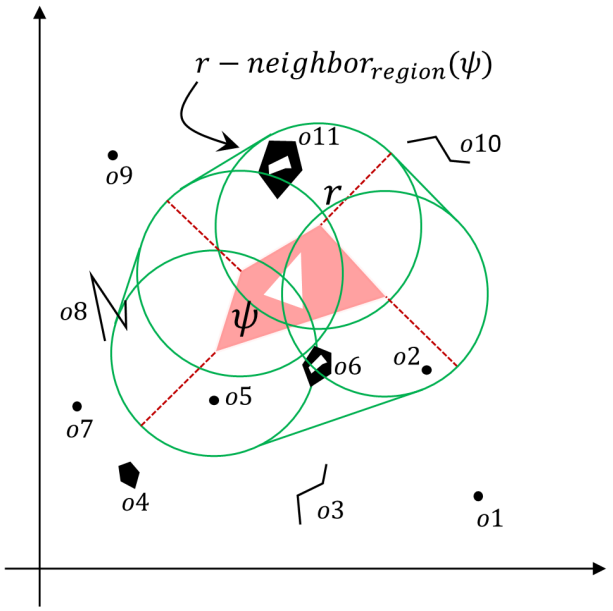


FIGURE 4. r-neighbor$_{region}(\psi)$ and r-neighbors(ψ)

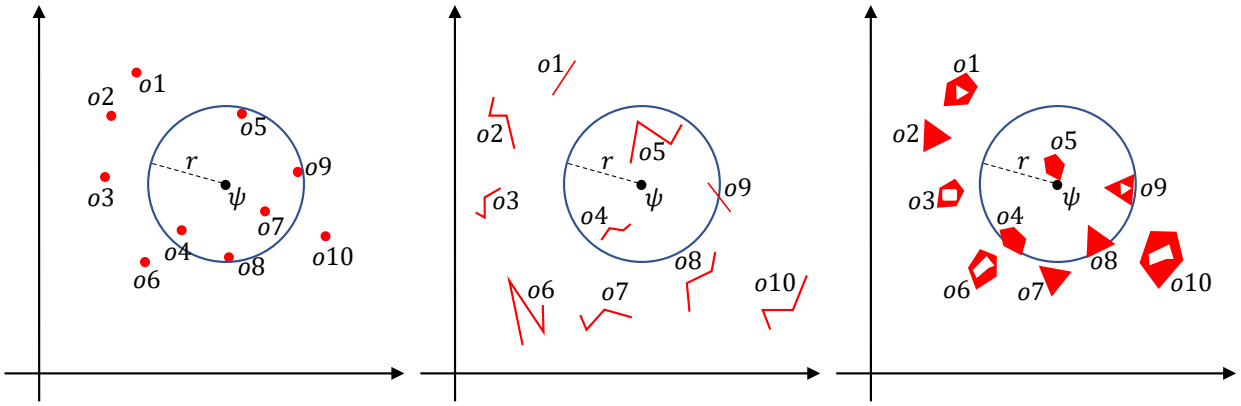
1) Point as a Query Object

Given Point ψ as a query object, Figure 5 shows the r-neighbors(ψ) computation for the spatial objects Point, LineString, and Polygon. In case of Point, r-neighbors(ψ) is defined as the objects which lie within r distance of ψ. Namely, all the spatial objects overlapping the r-neighbor$_{region}(\psi)$ are r-neighbors(ψ). In the Figure 5, r-neighbor$_{region}(\psi)$ is given by a circle around ψ due to the use of 2D Euclidean distance. In Figure 5(a), Points $o4$, $o5$, $o7$, $o8$, and $o9$, in Figure 5(b), LineStrings $o4$, $o5$, and $o9$, whereas in Figure 5(c) Polygons $o4$, $o5$, $o8$, and $o9$ are r-neighbors(ψ).

2) LineString or Polygon as a Query Object

Given LineString or Polygon ψ as a query object, Figures 6 and 7, respectively, show the r-neighbors(ψ) computation for the geometry types Point, LineString, and Polygon. Just like Point, r-neighbors(ψ) for LineString or Polygon query objects is defined as the objects which lie within r distance of ψ. Precisely, r-neighbors(ψ) is the set of spatial objects overlapping the region r-neighbor$_{region}(\psi)$. However, in contrast to Point query object, region r-neighbor$_{region}(\psi)$ of LineString or Polygon query object is not a circle, as the region is constructed by considering all the points of the LineString or Polygon query object. As for the LineString query object, in Figure 6(a), Points $o4$, $o5$, $o7$, $o8$, $o9$, and $o10$, in Figure 6(b), LineStrings $o4$, $o5$, $o8$, and $o9$, whereas in Figure 6(c) Polygons $o4$, $o5$, $o7$, $o8$, $o9$, and $o10$ are r-neighbors(ψ). Similarly for Polygon query object, in Figure 7(a), Points $o4$, $o5$, $o6$, $o7$, $o8$, $o9$, and $o10$, in Figure 7(b),

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

FIGURE 5. r-neighbors(ψ) evaluation for the query type *Point*

LineStrings $o1$, $o4$, $o5$, $o7$, $o8$, and $o9$, whereas in Figure 7(c) Polygons $o3$, $o4$, $o5$, $o6$, $o7$, $o8$, $o9$, and $o10$, are r-neighbors(ψ).

VI. SPATIAL STREAM INDEXING

Spatial data index structures can be classified into two broad categories: 1) Tree-based, and 2) Grid-based. Tree-based spatial indexes like R-tree, Quad-tree and KDB-tree can significantly speed-up the spatial query processing; however, their maintenance/restructuring cost is high specially in the presence of heavy updates (insertions and deletions) [33]. On the other hand, grid-based indexes enable fast updates. However, they cannot answer queries as efficiently as tree-based indexes [34] [35].

Since GeoFlink is a distributed spatial data stream management system, an index structure which can work efficiently in distributed environment in the presence of heavy updates is desirable. Considering the distributed processing in GeoFlink, where all the nodes work independently, the index must be maintained on each cluster node corresponding to the data it receives. Tree index structure is data dependent, thus resulting in different maintenance cost for each cluster node depending on the data it receives. Whereas, grid index does not require any maintenance. To this end, grid-based index seems to be a natural choice for GeoFlink.

One problem with the grid-based index is that it requires data boundaries for its construction. Since a stream is dynamic and infinite, its boundaries cannot be known in advance; however, it can be estimated based on the geographical location of the sensors generating the data stream in case of fixed sensors or it can be estimated by utilizing the loose bounds of the area or city of the objects, if the data stream is being generated by moving objects. For instance, assuming that we are aware of the city of stream source(s), we can use its geographical boundaries for the gird-index creation.

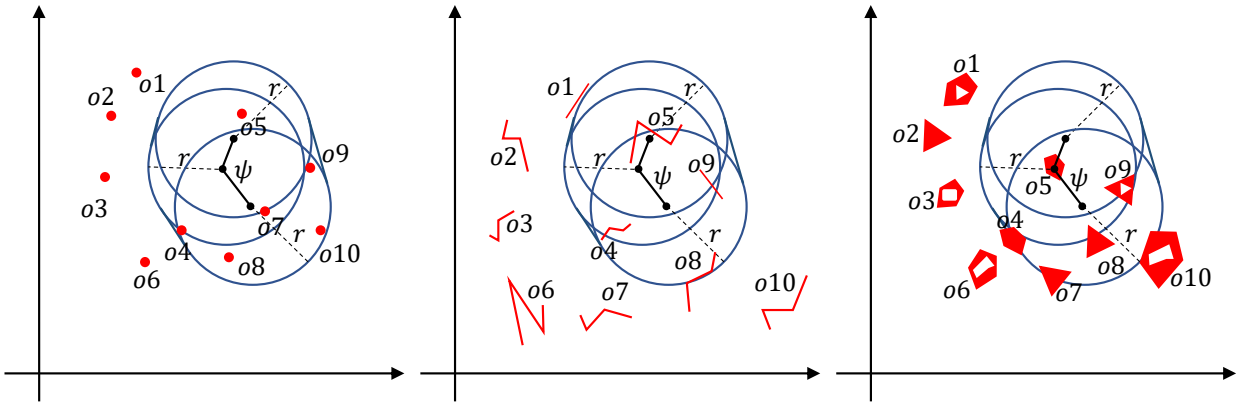
A. GEOFLINK GRID INDEX

The grid index used in this work is partially borrowed from our previous work [10], where it was mainly used for spatial Point object. However, in this work, we extend the grid index for other spatial objects discussed in Section IV. For brevity, we summarize it from scratch here.

The grid index [36] in this work is aimed at filtering or pruning objects during spatial queries' execution. Furthermore it helps in the near-uniform distribution of spatial objects across distributed cluster nodes. In GeoFlink, the grid index (G) is constructed by partitioning a 2D rectangular space, given by $(MinX, MinY), (MaxX, MaxY)$ $(MaxX - MinX = MaxY - MinY)$, into square shaped cells of length l, as shown in Figure 8. Here we assume that G's boundary is known, which can be estimated through data source location. Let $c_{x,y} \in G$ denotes a grid cell with indices x and y, respectively. We will use c instead of $c_{x,y}$ when it is clear from the context. Each cell $c \in G$ is identified by its unique cell ID (key) obtained by concatenating its x and y binary indices. Figure 8 shows a grid structure with a cell $c_{x,y}$ and its unique key. The $Gauranteed_{cells}(\psi)$ and $Candidate_{cells}(\psi)$ in Figure 8 are discussed in Section VI-B.

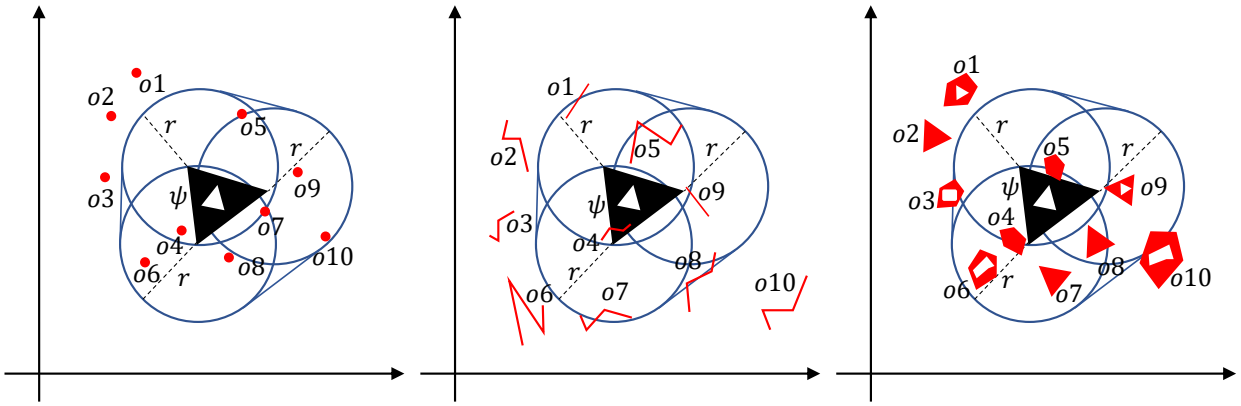
Within GeoFlink, each stream object is assigned one or more cell IDs ($keys$) on its arrival, depending on the G cells it belongs. A spatial object ψ belongs to a set of cells C if it overlaps partially or completely with $c \in C$. In this work, we assume that a spatial Point belongs to only one cell, whereas other spatial objects can belong to multiple cells depending on their sizes and positions. Hence, a single key is assigned to a Point whereas a set of $keys$ may need to be assigned to other spatial objects. Let s denotes a stream (S) tuple containing Point object, then its coordinates are given by $s.x$ and $s.y$. Given the grid boundary $(MinX, MinY), (MaxX, MaxY)$ and grid size g, l is given by $l = \frac{MaxX - MinX}{g}$, and the key of a $s \in S$ is obtained as $xIndex = \lfloor \frac{s.x - MinX}{l} \rfloor$, $yIndex = \lfloor \frac{s.y - MinY}{l} \rfloor$, and $s.key = xIndex \odot yIndex$, where $xIndex$ and $yIndex$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

FIGURE 6. r-neighbors(ψ) evaluation for the query type *LineString*



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

FIGURE 7. r-neighbors(ψ) evaluation for the query type *Polygon*

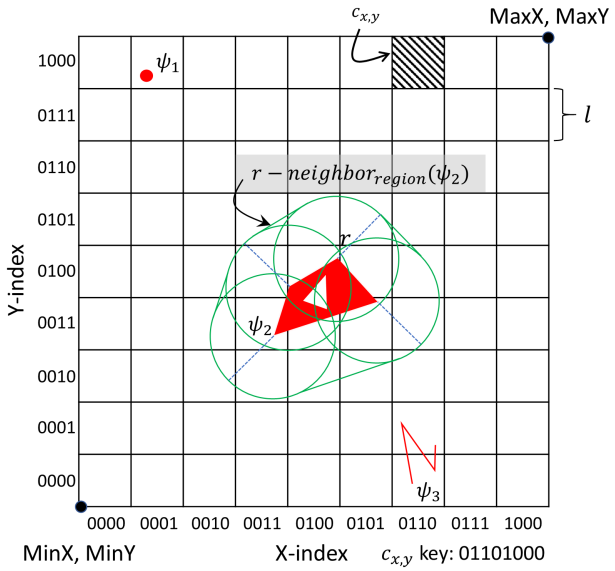


FIGURE 8. Grid index

are fixed length binary indices of bit length n and $\odot$ denotes a concatenation operator. For instance, consider the grid index shown in Figure 8. Let G is given by $(MinX, MinY) = (0, 0)$, $(MaxX, MaxY) = (90, 90)$ and $g = 9$, and let $n = 4$, then the keys of cell $c_{x,y}$, and Point (ψ_1), Polygon (ψ_2), and LineString (ψ_3) objects in the grid are given as follows: $c_{x,y}.key = 01101000$, $\psi_1.key = \{00011000\}$, $\psi_2.keys = \{00110011, 01000011, 01010011, 01010100, 01000100\}$, and $\psi_3.keys = \{01100000, 01100001\}$.

The grid-based index used in this work is logical, that is, it only assigns $keys$ to the incoming streaming objects. Besides, no physical data structure is needed, hence no update is required when an object expires or an updated object location is received. This makes our grid index fast and memory efficient.

Cost Analysis: The GeoFlink's grid index is logical, i.e., no structure is maintained in memory. Thus, there is no insert, search or delete cost. Moreover, being logical the index does not occupy any memory space. In GeoFlink, each stream tuple is assigned a grid cell key or a set of $keys$ based on its coordinates, which is used for its neighborhood computation,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

pruning and distribution across the cluster nodes. Thus, $keys$ computation using stream tuple object's coordinates is the only grid indexing and maintenance cost which is discussed above. Besides, Section VI-B details the use of neighborhood computation.

In GeoFlink, given g and index boundary $(MinX, MinY)$, $(MaxX, MaxY)$, a grid index is constructed via *UniformGrid* class.

```
UniformGrid G = new UniformGrid(g,
                                MinX, MaxX, MinY, MaxY);
```

where g=*50* generates a grid of size 50x50 cells, with the bottom-left $(MinX, MinY)$ and top-right $(MaxX, MaxY)$ coordinates, respectively.

B. GRID INDEX AND R-NEIGHBORS COMPUTATION

This section discusses, given a spatial query object ψ, how the proposed grid index is used for efficient query computation in GeoFlink. In particular, r-neighbors(ψ) computation. One traditional and a very effective approach to reduce query computation cost is to prune out the objects which cannot be part of the query output without using distance computation, because it can be costly specially for complex spatial objects. In the following, we discuss how our grid index is used to identify r-neighbors(ψ) and prune non r-neighbors(ψ).

Firstly, we need to identify C_ψ (set of cells containing the spatial object ψ). This is done by obtaining C_{bbox_ψ} (set of cells containing $bbox_\psi$ (bounding box of ψ)) and then computing C_ψ by identifying the cells in C_{bbox_ψ} overlapping ψ. The set C_{bbox_ψ} is obtained by fetching ψ's extreme coordinates to get its bounding box ($bbox_\psi$) and using it to identify the cells which lie within $bbox_\psi$. Figure 9(a) shows a query Polygon ψ with its $bbox_\psi$ and C_{bbox_ψ} (the gray cells). After obtaining $C_\psi \subset G$, the grid is divided into non-overlapping regions Guaranteed$_{cells}(\psi)$, Candidate$_{cells}(\psi)$, and Non-neighboring$_{cells}(\psi)$, as follows:

Definition 7 (Guaranteed$_{cells}(\psi)$)**.** A set of cells such that the objects in them are guaranteed to be r-neighbor(ψ).
Guaranteed$_{cells}(\psi) = \cup_{c_{x,y} \in C_\psi}\{c_{u,v} | x - \alpha \leq u \leq x + \alpha, y - \alpha \leq v \leq y + \alpha\}$, where $\alpha = \lfloor \frac{r}{l\sqrt{2}} \rfloor - 1$

Definition 8 (Candidate$_{cells}(\psi)$)**.** A set of cells such that the objects in them may or may not be r-neighbor(ψ). Hence, require (distance) evaluation.
Candidate$_{cells}(\psi) = \cup_{c_{x,y} \in C_\psi}\{c_{u,v} | x - \beta \leq u \leq x + \beta, y - \beta \leq v \leq y + \beta\}$ − Guaranteed$_{cells}(\psi)$, where $\beta = \lceil \frac{r}{l} \rceil$.

Definition 9 (Non-neighboring$_{cells}(\psi)$)**.** A set of cells such that the objects in them cannot be r-neighbor(ψ).
Non-neighboring$_{cells}(\psi)$ = $\{c_{x,y} \mid c_{x,y} \notin$ Guaranteed$_{cells}(\psi) \wedge c_{x,y} \notin$ Candidate$_{cells}(\psi)\}$

Algorithm 1 shows the steps to compute Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$ given ψ, r, and G. Given the definitions Guaranteed$_{cells}(\psi)$, Candidate$_{cells}(\psi)$, and Non-neighboring$_{cells}(\psi)$, Guaranteed-, Candidate-, and Non-neighbors of a spatial object ψ can be given as follows:

Algorithm 1 Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$ computation

Input: ψ: Spatial object, r: Query distance, G: Spatial grid index
Output: Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$

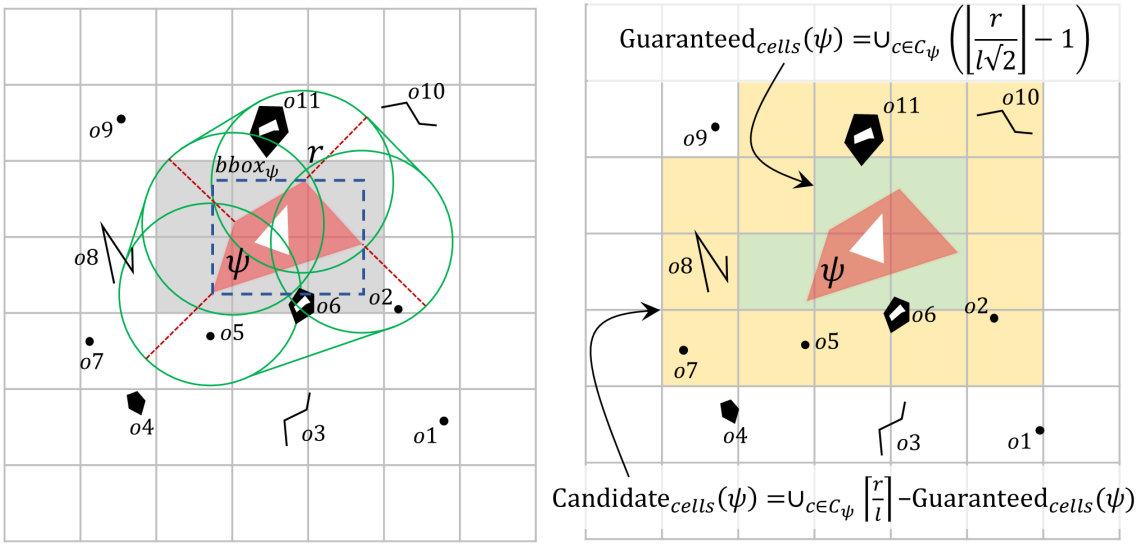
1: Obtain $bbox_\psi$ by identifying ψ's extreme x and y coordinates of
2: Compute C_{bbox_ψ}; C_{bbox_ψ} = Cells overlapping $bbox_\psi$. Since grid cells' boundaries are known, C_{bbox_ψ} can be obtained by fetching the grid cells overlapping $bbox_\psi$.
3: Compute C_ψ; $C_\psi \subset C_{bbox_\psi}$ and is obtained by identifying the cells in C_{bbox_ψ} overlapping ψ.
4: **for each** $c_{x,y} \in C_\psi$ **do**
5: Guaranteed$_{cells}(c_{x,y})$ = $\{c_{u,v} | x - \alpha \leq u \leq x + \alpha, y - \alpha \leq v \leq y + \alpha\}$, where $\alpha = \lfloor \frac{r}{l\sqrt{2}} \rfloor - 1$
6: Candidate$_{cells}(c_{x,y})$ = $\{c_{u,v} | x - \beta \leq u \leq x + \beta, y - \beta \leq v \leq y + \beta\}$ − Guaranteed$_{cells}(c_{x,y})$, where $\beta = \lceil \frac{r}{l} \rceil$
7: Guaranteed$_{cells}(\psi)$ = Guaranteed$_{cells}(\psi)$ ∪ Guaranteed$_{cells}(c_{x,y})$
8: Candidate$_{cells}(\psi)$ = Candidate$_{cells}(\psi)$ ∪ Candidate$_{cells}(c_{x,y})$
9: **end for**
10: **return** Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$

- Guaranteed-neighbors(ψ) = {object $\psi' | \psi'$ overlaps Guaranteed$_{cells}(\psi)$}
- Candidate-neighbors(ψ) = {object $\psi' | \psi'$ overlaps Candidate$_{cells}(\psi) \wedge \psi' \notin$ Guaranteed-neighbors(ψ)}
- Non-neighbors(ψ) = {object $\psi' | \psi' \notin$ Guaranteed-neighbors(ψ) $\wedge \psi' \notin$ Candidate-neighbors(ψ)}

Example 2. Let ψ denotes a spatial Polygon object (red colored) in 2D space as shown in Figure 9(a). Given distance r and assuming that ψ denotes a query object, the r-neighbor$_{region}(\psi)$ is given by green boundary around ψ. In addition to ψ, there exist 11 other spatial objects, i.e., $o1 \sim o11$. Furthermore, the Figure 9(a) shows grid index, where the gray cells denote C_{bbox}. The query object cells are denoted by C_ψ. Namely, C_ψ is a set of cells which overlap ψ. Using $c \in C_\psi$ we compute Guaranteed$_{cells}(\psi)$, Candidate$_{cells}(\psi)$, and Non-neighboring$_{cells}(\psi)$ as shown in Figure 9(b). Given the guaranteed-, candidate-, and non-neighboring cells, objects $o6$ and $o11$ are classified as guaranteed, $o2$, $o5$, $o7$, $o8$, and $o10$ as candidate, whereas $o1$, $o3$, $o4$, and $o9$ as non- r-neighbors(ψ).

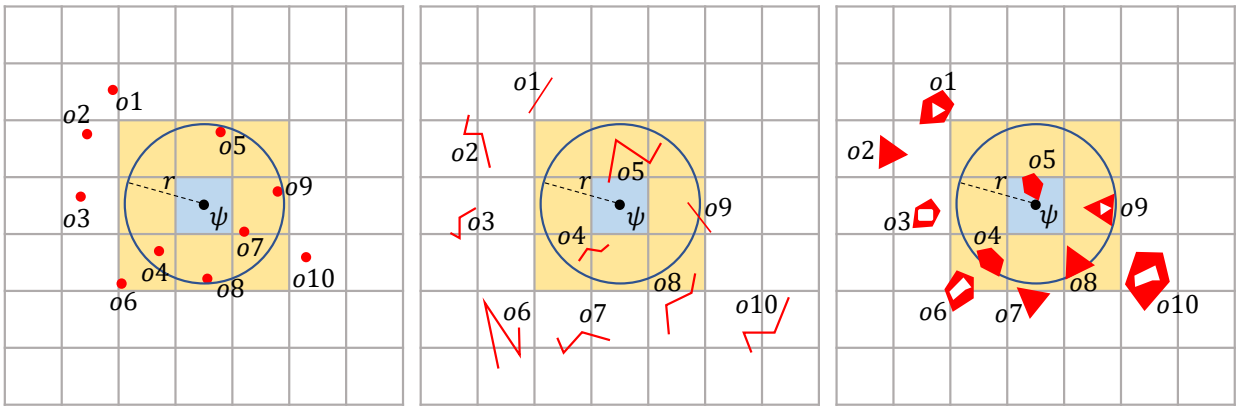
Figures 10, 11, and 12 show the Guaranteed$_{cells}(\psi)$, Candidate$_{cells}(\psi)$ and Non-neighboring$_{cells}(\psi)$ for the query objects Point, LineString and Polygon, respectively. In the figures, query objects are represented by ψ while other objects are represented by $o1, o2, ..., o10$. In case of LineString and Polygon query objects, respective bounding boxes are used to identify the cells containing them. The blue cells represent Guaranteed$_{cells}(\psi)$, the yellow

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



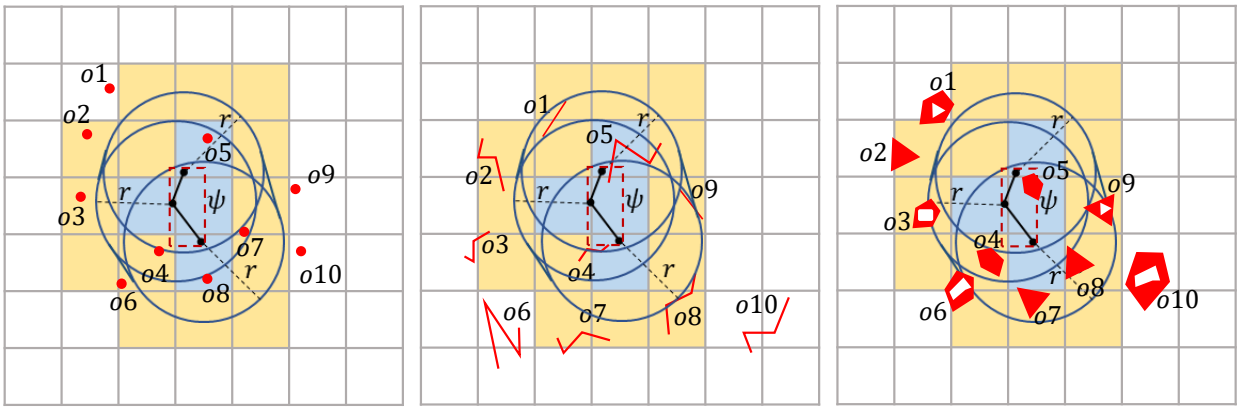
(a) Grid index and r-neighbor$_{region}(\psi)$ (b) Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$

FIGURE 9. Guaranteed-, Candidate- and Non-neighboring cells computation



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

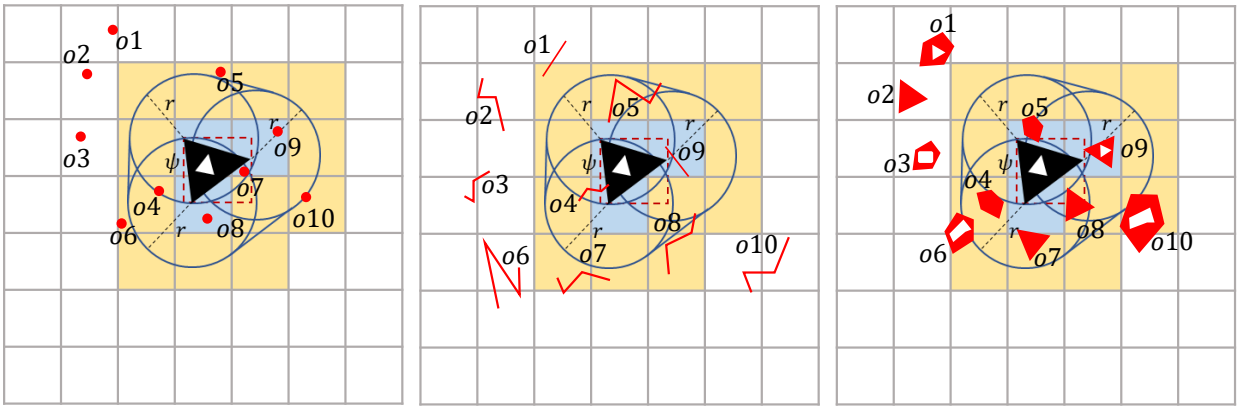
FIGURE 10. Point Grid



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

FIGURE 11. LineString Grid

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Geometry type: Point (b) Geometry type: LineString (c) Geometry type: Polygon

FIGURE 12. Polygon Grid

cells represent Candidate$_{cells}(\psi)$, while the white cells represent Non-neighboring$_{cells}(\psi)$. Please note that all $c \in$ Guaranteed$_{cells}(\psi)$ overlap the r distance of query geometry ψ, thus, all the objects in Guaranteed$_{cells}(\psi)$ are r-neighbors(ψ). Objects which overlap Candidate$_{cells}(\psi)$ may or may not be in r-neighbors(ψ), thus require actual distance computation. While objects which entirely lie in Non-neighboring$_{cells}(\psi)$ can be safely pruned.

VII. SPATIAL CONTINUOUS QUERIES

This section presents three basic spatial continuous operators/queries proposed in this work, i.e., spatial range, spatial kNN and spatial join. These queries are required by most of the spatial data processing and analysis applications. The spatial continuous queries are executed on continuous data streams. Each GeoFlink query has two variants, i.e., real-time and window-based.

- **Real-time:** Real-time query is triggered with the arrival of new stream tuples. Precisely, as a new tuple is received by GeoFlink, it is processed by real-time query and a corresponding output (if any) is generated.
- **Window-based:** Triggering of window-based query is based on window size and window slide step. The window-based query performs computation on all the spatial objects in the window and generates output (if any) corresponding to all the window contents. The query output is generated every slide step (W_s) in case of sliding window or every window size (W_n) in case of tumbling window.

In GeoFlink, a query can be configured as a real-time or window-based by utilizing *QueryConfiguration* class and providing it appropriate parameters. In the following code snippets, we will use *realTimeConf* and *windowBasedConf* for the real-time and window-based queries configuration, respectively.

```
// Real-time Query Configuration
int omegaDuration = 1; // time unit: seconds

QueryConfiguration realTimeConf = new
QueryConfiguration(QueryType.RealTime);
realTimeConf.setWindowSize(omegaDuration);

// Window-based Query Configuration
long windowSize = 1000000;
long windowSlideStep = 1000;

QueryConfiguration windowBasedConf = new
QueryConfiguration(QueryType.WindowBased);
windowBasedConf.setWindowSize(W_n);
windowBasedConf.setSlideStep(W_s);
```

Code 2. A GeoFlink (Java) code for query type configuration

Table 1 lists the notations used in the rest of the manuscript.

Notation	Definition
r	Query distance
ψ	Spatial object
S	Stream of spatial objects
s	Stream tuple
G	Grid index
g	Grid size in number of cells
key	Grid cell identifier
W_n	Window size
W_s	Window slide step
ω	Recency duration in seconds

TABLE 1. Commonly used notations

A. SPATIAL RANGE QUERY

This query comes in handy when a user wants to fetch spatial objects within certain distance of query objects.

Definition 10 (Spatial Range Query)**.** Given S, ψ, and r, range query returns the r-neighbors(ψ) in S continuously.

Namely, spatial range query returns all the $s \in S$ that lie within r distance of ψ.

The spatial range query makes use of grid-index for efficient computation. Using the proposed approach, the $s \in S$ objects which cannot be part of the query result are pruned out at an early stage, thus, reducing the number of distance

Algorithm 2 Spatial Range Query - Real-time

Input: S_{in}: Spatial data stream, ψ: Query object, r: Query distance, G: Spatial grid index
Output: S_{out}: Continuous spatial stream of r-neighbors(ψ)

1: Compute Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$ using ψ, r and G (Algorithm 1)
{Filter Phase}
2: **if** $(s \in S_{in})$ lies within or overlaps the Guaranteed$_{cells}(\psi)$ **then**
3: s is a guaranteed-neighbor and is added to S_{out}
4: **else if** $(s \in S_{in})$ lies within or overlaps the Candidate$_{cells}(\psi)$ **then**
5: s is a candidate-neighbor and is added to S_c {S_c denotes a filtered stream containing only candidate neighbors}
6: **end if**
{Shuffle Phase}
7: Shuffle $s \in S_c$ based on $s.key$ to balance the load across the distributed cluster nodes
{Refine Phase}
8: **if** $(s \in S_c)$ is in r-neighbors(ψ) **then**
9: Add s to S_{out} {r-neighbours(ψ) is computed using distance function}
10: **end if**
11: **return** S_{out}

computations and the query processing cost. Algorithm 2 shows the key phases of the distributed spatial range query, i.e., 1) Filter, 2) Shuffle, and 3) Refine. The algorithm is executed in a distributed fashion on a cluster of nodes. Each node receives data stream and executes *Filter Phase* (Lines 3 $\sim$ 7) independently. This phase directs guaranteed-neighbors to the output, candidate-neighbors to the next phase and prunes out non-neighbors based on their $keys$. For the spatial tuples containing a set of $keys$ with size > 1, if any of its key belongs to Guaranteed$_{cells}(\psi)$, it is a guaranteed-neighbor and is directed to the output. If none of its $keys$ belongs to Guaranteed$_{cells}(\psi)$, it is checked for candidate neighborhood in the similar fashion. The filter phase is followed by the *Shuffle Phase* which logically re-distributes the filtered stream tuples across cluster nodes based on their $keys$. Namely, all tuples with the same key are assigned to the same partition (Line 8). Finally, the *Refine Phase* evaluates the candidate stream tuples using spatial distance function and directs r-neighbor(ψ) to the output. Please note that the refine phase is executed in a distributed fashion, where each node receives one or more logical streams partitions from the shuffle phase.

Algorithm 3 presents the window-based version of the range query. The only difference between the real-time and window-based range query algorithms is the refine phase. In the real-time range query, the refine phase generates output continuously. Whereas, in the window-based range query, the output is generated periodically every W_s time units corresponding to the window size W_n.

Algorithm 3 Spatial Range Query - Window-based

Input: S_{in}: Spatial data stream, ψ: Query object, r: Query distance, W_n: Window size, W_s: Window step size, G: Spatial grid index
Output: S_{out}: Periodic spatial stream of r-neighbors(ψ)

1: Compute C_ψ {C_ψ denotes the cell(s) containing ψ}
2: Compute Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$ using ψ, r and G (Algorithm 1)
{Filter Phase}
3: **if** $(s \in S_{in})$ lies within or overlaps the Guaranteed$_{cells}(\psi)$ **then**
4: s is a guaranteed-neighbor and is added to S_{out}
5: **else if** $(s \in S_{in})$ lies within or overlaps the Candidate$_{cells}(\psi)$ **then**
6: s is a candidate-neighbor and is added to S_c {S_c denotes a filtered stream containing only candidate neighbors}
7: **end if**
{Shuffle Phase}
8: Shuffle $s \in S_c$ based on $s.key$ to balance the load across the distributed cluster nodes
{Refine Phase}
9: **for each** $(s \in S_c)$ in W {W denotes a window of size W_n and step W_s} **do**
10: **if** $(s \in S_c)$ is in r-neighbors(ψ) **then**
11: Add s to S_{out} {r-neighbours(ψ) is computed using distance function}
12: **end if**
13: **end for**
14: **return** S_{out}

To execute a real-time or window-based range query via GeoFlink, *run* method of one of the classes listed in Table 2 is used with appropriate query configuration discussed in Code 2.

Data stream	Query object	Class
Point	Point	PointPointRangeQuery
Point	LineString	PointLineStringRangeQuery
Point	Polygon	PointPolygonRangeQuery
LineString	Point	LineStringPointRangeQuery
LineString	LineString	LineStringLineStringRangeQuery
LineString	Polygon	LineStringPolygonRangeQuery
Polygon	Point	PolygonPointRangeQuery
Polygon	LineString	PolygonLineStringRangeQuery
Polygon	Polygon	PolygonPolygonRangeQuery

TABLE 2. Range query classes for different spatial objects

The code snippet 3 shows the registration of a range query on Point data stream and Point query object.

```
//queryConf could be realTime or windowBased
PointPointRangeQuery(queryConf, G).run(S, ψ, r);
```

Code 3. A GeoFlink (Java) class for Spatial Range Query

B. SPATIAL KNN QUERY

This query is useful in fetching k nearest spatial objects of a query object.

Definition 11 (Spatial kNN Query)**.** Given S, ψ, r and a positive integer k, kNN query returns the k nearest r-neighbors(ψ) in S. If less than k neighbors exist, all the r-neighbors(ψ) are returned.

In addition to the parameters discussed in Definition 11, real-time kNN query takes an additional parameter *recency duration* (ω). The real-time kNN query continuously outputs the k nearest r-neighbors(ψ) in S, where $s \in S$ must have arrived in the last ω duration. In contrast, window-based kNN takes two additional parameters, i.e., window size (W_n) and window slide step (W_s) and periodically outputs the k nearest r-neighbors(ψ) in S, every W_s time units, corresponding to window of size W_n. In practice, ω is much smaller than W_n and thus results in near real-time kNN.

Algorithm 4 Spatial kNN Query - Real-time

Input: S_{in}: Spatial data stream, ψ: Query object, k: Number of desired nearest neighbors, ω: Recency duration, G: Spatial grid index

Output: S_{out} : Continuous spatial stream of the k nearest r-neighbors(ψ) within ω duration

1: Compute C_ψ {C_ψ denotes the cell(s) containing ψ}
2: Compute Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$ using ψ, r and G (Algorithm 1)
{Filter Phase}
3: **if** $(s \in S_{in})$ lies within or overlaps the Guaranteed$_{cells}(\psi) \vee (s \in S_{in})$ lies within or overlaps the Candidate$_{cells}(\psi)$ **then**
4: s is a candidate kNN and is added to S_c {S_c denotes a filtered stream containing only candidate neighbors}
5: **end if**
{Shuffle Phase}
6: Shuffle $s \in S_c$ based on $s.key$ to balance the load across the distributed cluster nodes
{Refine Phase}
7: **for each** $(s \in S_c)$ in Ω {Ω denotes a window of size ω} **do**
8: **if** $(s \in S_c)$ is in r-neighbors(ψ) $\wedge$ $(s \in S_c)$ is a kNN **then**
9: Add s to S_{knn} {S_{knn} denotes stream of k nearest neighbors}
10: Update S_{knn} to keep the size of S_{knn} at most k
11: **end if**
12: **end for**
{Merge Phase}
13: Merge and sort $s \in S_{knn}$ from all the distributed nodes to obtain an integrated kNN and add the result to S_{out}
14: **return** S_{out}

Just like the range query, spatial kNN query makes use of grid index for efficient computation. Using the proposed approach, $s \in S$ which cannot be part of the query result are pruned out at an early stage, thus reducing the number of distance computations and the query processing cost. Algorithm 4 shows the key phases of the distributed spatial range query, i.e., 1) Filter, 2) Shuffle, 3) Refine, and 4) Merge. The algorithm is executed in a distributed fashion on a cluster of nodes. Each node receives data stream and executes *Filter Phase* (Lines 3 $\sim$ 5) independently. This phase prunes out non-neighbors and directs candidate-neighbors in the guaranteed and candidate neighboring layers to the next phase. This phase is followed by the *Shuffle Phase* which logically re-distributes the filtered stream tuples across cluster nodes based on their $keys$. Namely, all tuples with the same key are assigned to the same partition (Line 6). The *Refine Phase* evaluates the candidate stream tuples arrived in the last ω duration for kNN using distance function (Lines 7 $\sim$ 12). Since the refine phase is computed independently on distributed nodes, *Merge Phase* is needed to integrate kNNs, which integrates the kNNs from all the distributed nodes to obtain true kNNs.

Since the real-time and window-based kNN algorithms are very similar, kNN window based algorithm has been omitted. The main difference between the real-time and window-based kNN algorithms is the refine phase, where the window parameters W_n and W_s are used instead of recency parameter (ω) for the window size. In practice, ω is much smaller than W_n.

Data stream	Query object	Class
Point	Point	PointPointKNNQuery
Point	LineString	PointLineStringKNNQuery
Point	Polygon	PointPolygonKNNQuery
LineString	Point	LineStringPointKNNQuery
LineString	LineString	LineStringLineStringKNNQuery
LineString	Polygon	LineStringPolygonKNNQuery
Polygon	Point	PolygonPointKNNQuery
Polygon	LineString	PolygonLineStringKNNQuery
Polygon	Polygon	PolygonPolygonKNNQuery

TABLE 3. kNN query classes for different spatial objects

To execute a real-time or window-based kNN query via GeoFlink, *run* method of one of the classes listed in Table 3 is used with appropriate query configuration discussed in Code 2. The code snippet 4 shows the registration of a kNN query on Point data stream and Point query object.

```
//queryConf could be realTime or windowBased
PointPointKNNQuery(queryConf, G).run(S, ψ, r, k);
```

Code 4. A GeoFlink (Java) class for Spatial kNN Query

Please note that the output of the kNN query is a stream of sorted lists with respect to the distance from ψ, where each list consists of kNNs corresponding to the last ω duration or window size W_n.

C. SPATIAL JOIN QUERY

Spatial join is a useful operator where one stream is joined with another based on some query distance. Join is an expensive operator as it involves Cartesian product between the two streams.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

Definition 12 (Spatial Join Query)**.** Given two spatial streams $S1$ (Ordinary stream) and $S2$ (Query stream) and a distance r, a join query returns all pairs of objects (ψ_i, ψ_j), such that $\psi_i \in S1$ is in r-neighbors(ψ_j), where $\psi_j \in S2$.

In addition to the parameters discussed in Definition 12, real-time join query takes an additional parameter *recency duration* (ω). The real-time join query continuously outputs the spatial join pairs (ψ_i, ψ_j), where $\psi_i \in S1$ and $\psi_j \in S2$ must have arrived in the last ω duration. In contrast, window-based spatial join takes two additional parameters, i.e., window size (W_n) and window slide step (W_s) and periodically (every W_s time units) outputs the spatial join pairs (ψ_i, ψ_j), where $\psi_i \in S1$ and $\psi_j \in S2$ belong to window of size W_n. In practice, ω is much smaller than W_n and thus results in near real-time spatial join.

Spatial join is an expensive operation, where each tuple of query stream must be checked against every tuple of ordinary stream. However, this involves a large number of distance computations. Hence, we propose an efficient grid index based spatial join. According to the spatial join definition (Definition 12), given a query stream object ψ_j, the objects which lie within r-neighbors(ψ_j) can be part of join output. Thus, a query stream object needs to be joined with the objects which lie only within r distance of it rather than all the objects. This requires some pruning strategy. To achieve this in a distributed environment using grid index, where the objects belonging to the same key (cell) lands on the same logical computing node for join processing, we replicate each query stream object based on its Guaranteed$_{cells}(\psi)$ and Candidate$_{cells}(\psi)$. We then join the replicated query stream with other stream using key-based join, causing the ordinary and query stream objects corresponding to a particular cell to land on the same logical join operator and prune out the objects with no matching keys (i.e., cell IDs). Thus, resulting in an efficient grid-based spatial stream join.

Algorithm 5 shows the distributed spatial join algorithm consisting of the following three phases: 1) Replication phase, 2) Join phase, and 3) Filter phase. Let $S1$ and $S2$ denote an ordinary and a query stream, respectively. Assuming that C_{ψ_j} denotes a set of cells containing a query object ψ_j, then given r, the *Replication Phase* computes Guaranteed$_{cells}(\psi_j)$ and Candidate$_{cells}(\psi_j)$ for each $\psi_j \in S2$. Next, the $\psi_j \in S2$ are replicated in such a way that each replicated object is assigned $keys$ of the cells in Guaranteed$_{cells}(\psi_j)$ and Candidate$_{cells}(\psi_j)$. We denote the replicated query stream by $S2'$. Next, in the *Join Phase*, we make use of Apache Flink's key-based join transformation to join the two streams, i.e., $S1$ and $S2'$. The Flink's key-based join enables the tuples from the two streams with the same key to land on the same operator instance. This causes the join to be evaluated only between $q \in S2'$ and $p \in S1$ belonging to the cells in Guaranteed$_{cells}(\psi_j)$ and Candidate$_{cells}(\psi_j)$, while filtering out the non-neighbors of ψ_j. Since the $\psi_i \in S1$ corresponding to Guaranteed$_{cells}(\psi_j)$ are guaranteed to be part of the join output, they are sent to the output directly without distance evaluation. However, for $\psi_i \in S1$ corresponding to Candidate$_{cells}(\psi_j)$, distance evaluation is done to find if $\psi_i \in S1$ is in r-neighbors(ψ_j), where $\psi_j \in S2'$. The *Filter Phase* filters out the output stream to remove the duplicate pairs, if any.

Since the real-time and window-based join algorithms are very similar, window-based join algorithm has been omitted. The main difference between the real-time and window-based join algorithms is the join phase, where the window parameters W_n and W_s are used instead of recency parameter (ω) for the window size. In practice, ω is much smaller than W_n.

Data stream	Query object	Class
Point	Point	PointPointJoinQuery
Point	LineString	PointLineStringJoinQuery
Point	Polygon	PointPolygonJoinQuery
LineString	Point	LineStringPointJoinQuery
LineString	LineString	LineStringLineStringJoinQuery
LineString	Polygon	LineStringPolygonJoinQuery
Polygon	Point	PolygonPointJoinQuery
Polygon	LineString	PolygonLineStringJoinQuery
Polygon	Polygon	PolygonPolygonJoinQuery

TABLE 4. Join query classes for different spatial objects

To execute a real-time or window-based join query via GeoFlink, *run* method of one of the classes listed in Table 4 is used with appropriate query configuration discussed in Code 2. The code snippet 5 shows the registration of a join query on Point data stream and Point query stream.

```
//queryConf could be realTime or windowBased
PointPointJoinQueryqueryConf, G, G_{ψ_j}
                              .run(S_1, S_2, r);
```

Code 5. A GeoFlink (Java) class for Spatial Join Query

Please note that the output of the join query is a stream of pairs (ψ_i, ψ_j), where $\psi_i \in S1$ and $\psi_j \in S2$ such that ψ_i is r-neighbors(ψ_j).

GeoFlink supports 6 types of spatial objects. Thus, we provide overloaded methods of each query to support these spatial objects as input stream and query object/stream. Table 5 lists the geometry combinations supported by GeoFlink for each operator.

Query Object/Stream	Input Stream
Point/MultiPoint	Point/MultiPoint
Point/MultiPoint	LineString/MultiLineString
Point/MultiPoint	Polygon/MultiPolygon
LineString/MultiLineString	Point/MultiPoint
LineString/MultiLineString	LineString/MultiLineString
LineString/MultiLineString	Polygon/MultiPolygon
Polygon/MultiPolygon	Point/MultiPoint
Polygon/MultiPolygon	LineString/MultiLineString
Polygon/MultiPolygon	Polygon/MultiPolygon

TABLE 5. Geometry combinations supported by GeoFlink for each operator

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

Algorithm 5 Spatial Join Query - Real-time

Input: $S1$: Spatial stream 1, $S2$: Spatial stream 2, r: Query distance, ω: Recency duration, G: Spatial grid index

Output: S_{out}: Continuous spatial stream of pairs (ψ_i, ψ_j), where $\psi_i \in S1$ and $\psi_j \in S2$ such that ψ_i is r-neighbor(ψ_j)

{Distributed $S2$ Replication Phase}

Compute C_{ψ_j} for each $\psi_j \in S2$ {C_{ψ_j} denotes the cell containing ψ_j}

Compute Guaranteed$_{cells}(\psi_j)$ and Candidate$_{cells}(\psi_j)$ using ψ_j, r and G (Algorithm 1)

for each $c \in$ Guaranteed$_{cells}(\psi_j) \vee c \in$ Candidate$_{cells}(\psi_j)$ **do**

  Add $\psi_j \in S2$ to $S2'$, such that $\psi_j.key$ is set to $c.key$

end for

{Distributed Join Phase}

Perform distributed key-based join between $S1$ and $S2'$, which directs all the tuples from $S1$ and $S2'$ with the same key to the same logical operator instance. This step filters out the $S1$ and $S2'$ tuples with non-matching keys.

for each $(\psi_i \in S1)$ in $\Omega \wedge (\psi_j \in S2')$ in Ω {Ω denotes a window of size ω} **do**

  if ψ_i is in r-neighbors(ψ_j) **then**

    Add (ψ_i, ψ_j) to S_c {r-neighbors(ψ_j) is computed using distance function}

  end if

end for

{Distributed Filter Phase}

Filter $(\psi_i, \psi_j) \in S_c$ to remove duplicate pairs and add the result to S_{out}

return S_{out}

VIII. GEOFLINK ARCHITECTURE

Figure 13 shows the GeoFlink architecture. Users can register queries to GeoFlink through a Java/Scala API and its output is available via a variety of sinks provided by Apache Flink. The GeoFlink architecture has two important layers: 1) Spatial Stream Layer and 2) Real-time Spatial Query Processing Layer.

A. SPATIAL STREAM LAYER

This layer is responsible for converting incoming data stream(s) into spatial data stream(s). Apache Flink treats spatial data stream as ordinary text stream, which may leads to its inefficient processing. GeoFlink converts it into spatial data stream of spatial objects. GeoFlink supports all the commonly used spatial objects including Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. Furthermore, this layer assigns grid cell IDs ($keys$) to the spatial objects for their efficient distribution and processing. GeoFlink's grid index is discussed in Section VI-A.

1) Spatial Objects Support

GeoFlink supports *GeoJSON*, *CSV* and *TSV* input stream formats from Apache Kafka. GeoFlink user needs to make

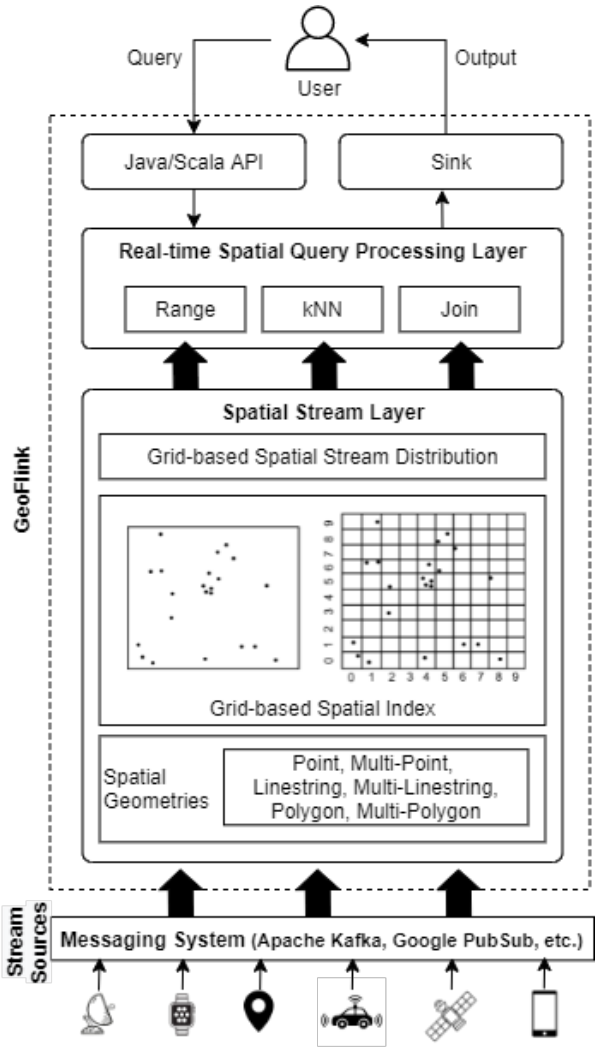


FIGURE 13. GeoFlink architecture

an appropriate Apache Kafka connection by specifying the kafka *topic name* and bootstrap server(s). Once the connection is established, users can construct spatial stream from the input streams by utilizing *PointStream*, *LineStringStream* or *PolygonStream* methods of the GeoFlink's *Deserialization* class.

2) Spatial Stream Partitioning

Uniform partitioning of data across distributed cluster nodes plays a vital role in efficient query processing. As discussed in Section III-A, Apache Flink *keyBy* transformation logically partitions a stream into disjoint partitions in such a way that all the tuples with the same key are assigned to the same partition or to the same operator instance.

To enable uniform data partitioning in GeoFlink, which takes into account spatial data proximity, grid index is used. GeoFlink assigns grid cell IDs ($keys$) to each incoming stream tuple based on its spatial location. Since all the spatially close tuples belong to a single grid cell, thus, are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

assigned the same key, which is used by the Flink's *keyBy* operator for stream distribution. It is good to have the number of keys greater than or equal to the amount of parallelism, to enable the Flink to distribute data uniformly.

It is worth mentioning that, GeoFlink receives distributed data streams from distributed messaging system, for instance, Apache Kafka [25]. To enable uniform distribution of incoming data stream across GeoFlink cluster nodes, right configuration is needed which is discussed in III-B.

B. REAL-TIME SPATIAL QUERY PROCESSING LAYER

This layer provides support for a number of basic spatial operators required by most of the spatial data processing and analysis applications. The supported operators include spatial range, kNN, and join over spatial data streams. The queries are continuous in nature, i.e., they generate continuous results on continuous spatial stream(s). Users can use Java or Scala to write the spatial queries or custom applications. This layer makes extensive use of the grid index for efficient queries' execution.

IX. EXPERIMENTAL EVALUATION

This section presents detailed experimental evaluation of GeoFlink. In the following, we discuss data streams and experimentation environment in Section IX-A and evaluation results in Section IX-B.

A. DATA STREAMS AND ENVIRONMENT

For GeoFlink evaluation, three real and four synthetic datasets are used. Microsoft TDrive (TDrive) [37], ATC shopping mall (ATC) [38], and NYC Buildings (NYCBuildings) [39] are real, whereas Gaussian, Random, Hotspot, and Multi-hotspot are synthetic datasets. Table 6 shows a quick summary of the datasets and Figure 14 shows the datasets' distribution.

Name	∼#Objects	Dataset Type
TDrive	17 million	Point
ATC	1.43 billion	Point
NYCBuildings	1 million	MultiPolygon and MultiLineString
Gaussian	100 million	Point, Polygon and LineString
Random	50 million	Point
Hotspot	50 million	Point
Multi-hotspot	50 million	Point

TABLE 6. Datasets used in experiments

TDrive is a trajectory dataset consisting of 10,357 taxi trajectories in the Beijing city of China. The dataset was collected during February 2 to 8, 2008. The dataset contains 17 million data points, with the trajectories covering a distance of around 9 million kilometres. Each data point consists of a taxi id, timestamp, longitude and latitude [37].

ATC is a pedestrians tracking dataset within a shopping center in the Osaka city of Japan. The dataset was collected between October 24, 2012 and November 29, 2013. The data collection was done every week on Wednesday and Sunday, from morning until evening (9:40-20:20). The dataset consists of 92 days in total and contains approximately 1.43 billion data points. Each data point consists of the following fields: timestamp, person id, position x [mm], position y [mm], position z (height) [mm], velocity [mm/s], angle of motion [rad], facing angle [rad] [38]. Table 6 shows a summary of the two datasets.

NYCBuildings dataset is obtained from NYC open data [39] and contains NYC buildings footprints. The buildings footprints represent the full perimeter outline of each building as viewed from directly above. Besides the perimeter, other useful attributes of this dataset include ground elevation at building base, roof height above ground elevation, construction year, and feature type. The buildings dataset consist of more than 1 million NYC buildings information including residential, commercial and government buildings. This dataset is used as a source of polygon and linestring data streams. Originally the dataset consists of polygon objects; the last point of each outer polygon is removed to convert it into linestring.

Gaussian, Random, Hotspot and Multi-hotspot dataset consists of streams of m distinct objects, where m is a user-defined parameter. Unless specified, m is set to 100 in the experiments. As for the Gaussian data stream, given user-defined number of points (vertices), the first m distinct objects are generated randomly at random locations. The next m objects are also generated randomly; however, their locations are Gaussian distributed, where mean is the last position or centroid of the corresponding object and variance is a user-defined parameter which is set to 0.5 for the dataset generation. The Random data stream follows random distribution; whereas Hotspot and Multi-hotspot follow Gaussian distribution with mean = {(75,75)}, variance = {(5,5)} and mean = {(10, 15), (55, 75), (90, 25)}, variance = {(3, 3), (6, 3), (3, 8)}, respectively.

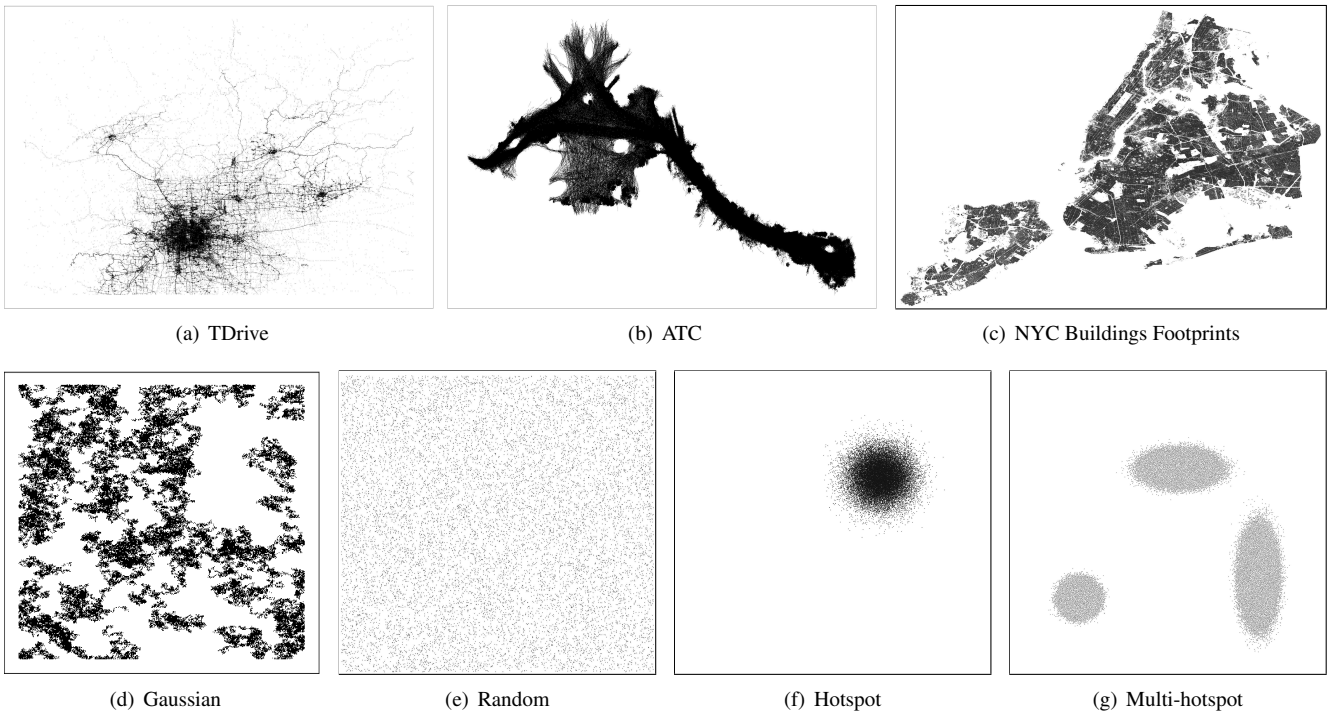
The synthetic Gaussian dataset generator can also be used to generate spatial trajectories and is available as an open source project [40].

For the experiments, a 4 nodes Apache Flink cluster with GeoFlink (1 Job Manager and 3 Task Managers (30 task slots)) and a 3 nodes Apache Kafka cluster (1 Zookeeper and 2 Broker Nodes) are used. The clusters are deployed on AIST AAIC cloud [41], where each VM has 128 GB memory and 20 CPU cores (Intel skylake 1800 MHz processor). All the VMs are operated by Ubuntu 16.04. The datasets are loaded into Apache Kafka [25] and are supplied as a distributed stream to GeoFlink cluster.

B. EVALUATION

This section presents detailed evaluation results. In particular, we evaluated GeoFlink's throughput and latency for the three spatial queries discussed in this manuscript. Unless otherwise specified, following default parameter values are used in the experiments: r = 0.005 (approx. 500 meters), W_n = 100 seconds, W_s = 50 seconds, g = 200, k = 50, ω = 1 second,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) TDrive (b) ATC (c) NYC Buildings Footprints

(d) Gaussian (e) Random (f) Hotspot (g) Multi-hotspot

FIGURE 14. Datasets for experiments

query stream arrival rate = 10 tuples per second, number of query objects = 50 (range query), 1 (kNN query).

- **Throughput:** The number of stream tuples processed by the system per second. It is obtained by dividing the total number of input stream tuples processed by the system with total processing time.
- **Latency:** The amount of time a stream tuple takes to be processed by the system after ingestion. It is calculated by assigning each stream object the system time on its arrival and subtracting it from the system time at its output generation.

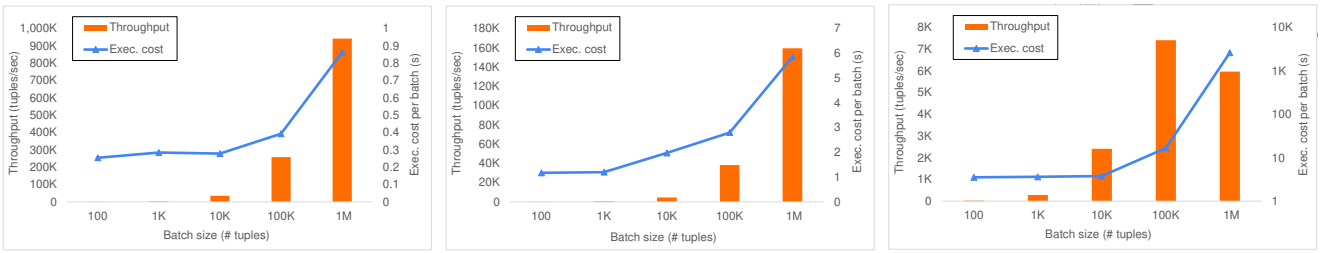
1) Throughput Comparison: GeoFlink, Apache Flink and Apache Spark Streaming

In this section, we compare GeoFlink throughput with Apache Flink and Apache Spark Streaming. In particular, we compare the throughput of real-time range (Point stream, a set of Polygon query objects), $k$nn (Point stream, a Polygon query object), and join (ordinary Point stream, query Point stream). In contrast to Apache Flink and Apache Spark Streaming, GeoFlink utilizes grid index; however, to keep the comparison fair, efforts are made to distribute the query processing uniformly across the cluster nodes.

Apache Spark is primarily developed for batch processing, thus, the internal architecture of it is different from Apache Flink. To process data streams, tuples are bundled as micro batches in contrast to Flink which processes each stream tuple separately. Therefore, we need to provide an additional parameter batch size for Spark Streaming. The appropriate batch size depends on the application. Larger batch sizes can improve throughput but increases latency, which is not desirable in many real-time streaming application. Since the target of this section is to compare the throughput, we performed preliminary experiments on Apache Spark Streaming with different batch sizes to identify the ideal batch size. For the preliminary experiments, TDrive dataset is used. Figure 15 compares the batch sizes, throughput and execution cost per batch for the real-time range, kNN and join queries. As can be observed, larger batch sizes can improve the query throughput but results in higher per batch processing cost. Higher batch processing cost causes the streaming tuples in the next batch to suffer from latency or processing delays. Furthermore, from the experiments we identify that for the very large batch size, i.e., 1 million tuples in case of the join query (Figure 15(c)), the throughput actually decreases instead of increasing. Furthermore, the execution cost per batch also increases exponentially. This is because of the fact that the join query involves Cartesian product of the two stream batches, which is an expensive operation. The larger the batch size, the larger the Cartesian product and higher the per batch processing cost. Keeping in view the preliminary experiment results on Apache Spark Streaming, we identify 100,000 tuples as an ideal batch size for our experiments. Thus, for the throughput comparison experiments batch size: 100,000 tuples (Spark Streaming only) is used.

Next, we present throughput comparison for the real-time range, kNN and join queries in Figures 17, 18, and 19, respectively. Figures 17(a), 17(b) and 17(c) show the throughput comparison of the real-time range query by varying the number of query Polygons for the TDrive, ATC and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Range query (b) kNN query (c) Join query

FIGURE 15. Spark batch size vs. throughput

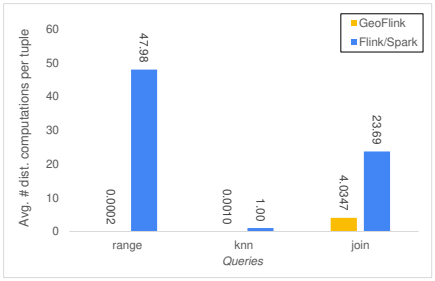


FIGURE 16. Effect of Grid-based Pruning

Synthetic datasets, respectively. From the figures, one can observe that GeoFlink performs better than Apache Flink and Apache Spark Streaming, i.e., results in higher throughput. This is due to the strong pruning capability of GeoFlink. Another thing to note in the figures is that the throughput decreases slightly with the increase in number of query Polygons, which is obvious as with the increase in the number query Polygons the required number of distance evaluations increases.

Figures 18(a), 18(b) and 18(c) show the same comparison for the kNN query by varying query distance. Here again the GeoFlink results in the maximum throughput, showing the advantage of GeoFlink pruning. Compared to the range query in Figure 17, Apache Spark Streaming results in the lower throughput than Apache Flink. We observed that as the query gets complex, the throughput of Apache Spark Streaming decreases. Furthermore, the throughput of all the approaches decreases with the increase in the query distance, as the query needs to check more query object's neighbors for the kNN computation.

In Figures 19(a), 19(b) and 19(c), we compare the throughputs of the real-time join query by varying the query stream arrival rate for the TDrive, ATC and Synthetic datasets, respectively. From the figures, the throughput of the Apache Flink and Apache Spark Streaming are significantly lower than GeoFlink because join is an expensive operation and in the absence of indexes, Cartesian product is computed between the two streams. Thus, resulting in quadratic distance computations and reduced throughput. On the other hand, GeoFlink's grid index helps in pruning out the streaming tuples which cannot be part of join output as can be observed from Figure 16, thus, resulting in higher throughput for GeoFlink. Another point to note in the figures is that the throughput decreases with the increase in query stream arrival rate. This is because, higher the query stream rate, larger the Cartesian product, resulting in reduced throughput. However, the decrease is not significant in case of GeoFlink, showing the power of GeoFlink indexing.

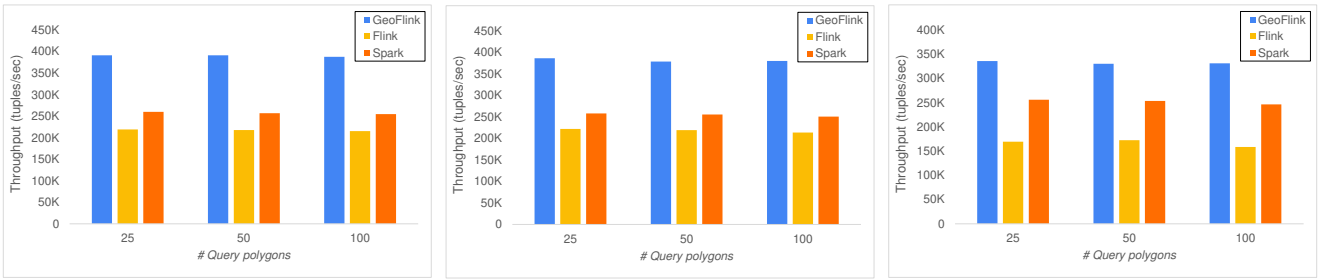
2) Skewed Data and Grid Index

In this section we evaluate the performance of the grid index on uniform and skewed datasets. For this sake, three Point datasets are used in this subsection (Sec. IX-B2), i.e., Random (Fig. 14(e)), Hotspot (Fig. 14(f)), and Multi-hotspot (Fig. 14(g)).

Figures 20 and 21 present the effect of grid size (g) on Random and skewed (Hotspot and Multi-hotspot) data distributions for real-time range, kNN and join queries, respectively. We observe low throughput for smaller g, i.e., 5 and 10, for all the queries. The throughput tends to become normal for grid sizes 50 and above. There are two reasons for the lower throughput: 1) Smaller g leads to ineffective grid-based pruning, 2) Smaller g results in imbalanced data distribution especially for skewed datasets.

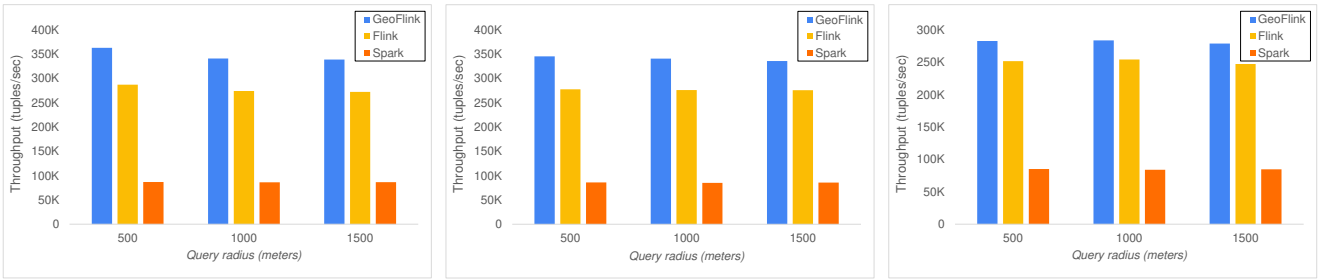
Figure 22 shows the task managers (TMs) data load for the real-time range query. The data load is computed at the last operator of the range query after gird-based pruning and shuffling (hashing), i.e., at the refine phase in Algorithm 2. It is computed after the processing of all the stream tuples of the respective datasets. One can observe imbalanced TM load for smaller g, i.e., 5, 10, 25 and 50, for the skewed datasets. The TM load becomes uniform for higher (finer) g because finer grid leads to smaller grid cell size and a large number of keys, thus, near-uniform hashing even for skewed datasets. Another point to note in Figure 22 is that the TM load decreases gradually from grid sizes 5 to 50 and then becomes steady. This is due to the reason that smaller g leads to ineffective grid-based pruning, thus, resulting in higher data load at the refine phase of Algorithm 2. From this we can conclude that grid index performs well for uniform and skewed data distributions if the g is chosen carefully.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



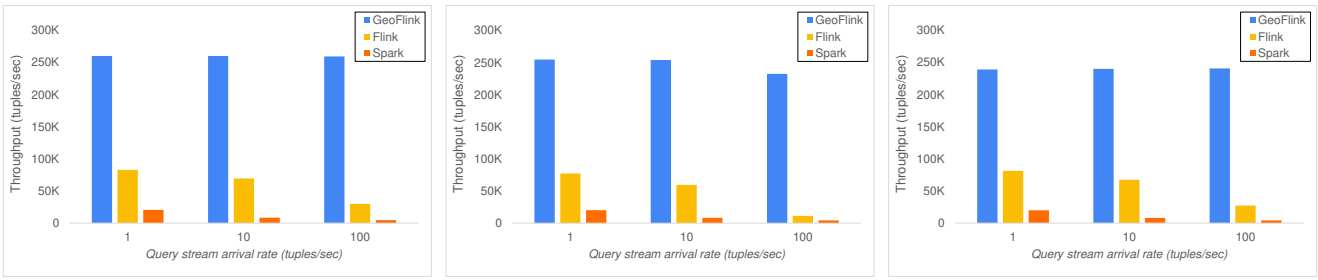
(a) TDrive dataset (b) ATC dataset (c) Synthetic dataset

FIGURE 17. Throughput comparison: Range query



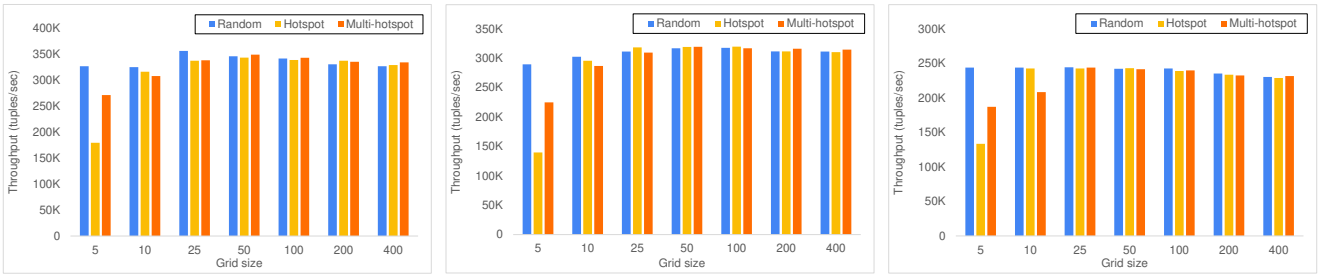
(a) TDrive dataset (b) ATC dataset (c) Synthetic dataset

FIGURE 18. Throughput comparison: KNN query



(a) TDrive dataset (b) ATC dataset (c) Synthetic dataset

FIGURE 19. Throughput comparison: Join query



(a) Range query (b) kNN query (c) Join query

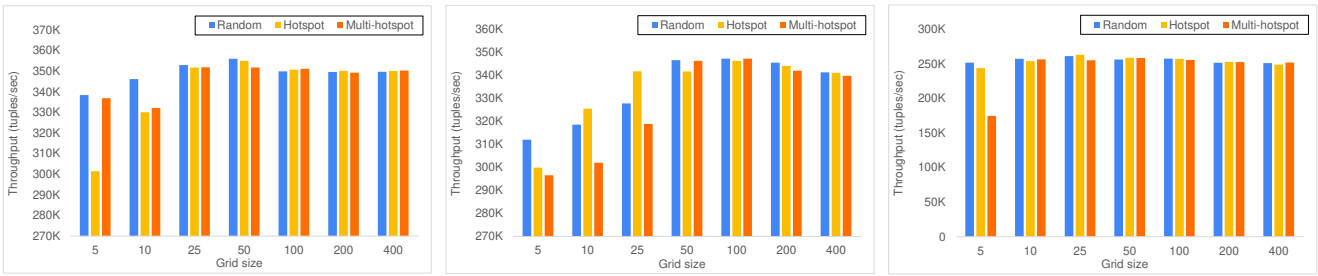
FIGURE 20. Num. of Queries: 1 (Smaller Grid size (larger cell size) results in lower grid-based pruning and imbalanced skewed data distribution across the distributed nodes; thus, resulting in lower throughput and vice versa)

3) GeoFlink Throughput

This subsection (Section IX-B3) presents a detailed throughput evaluation of GeoFlink queries. For these experiments, TDrive and NYCBuildings datasets are used for Point and LineString/Polygon objects, respectively.

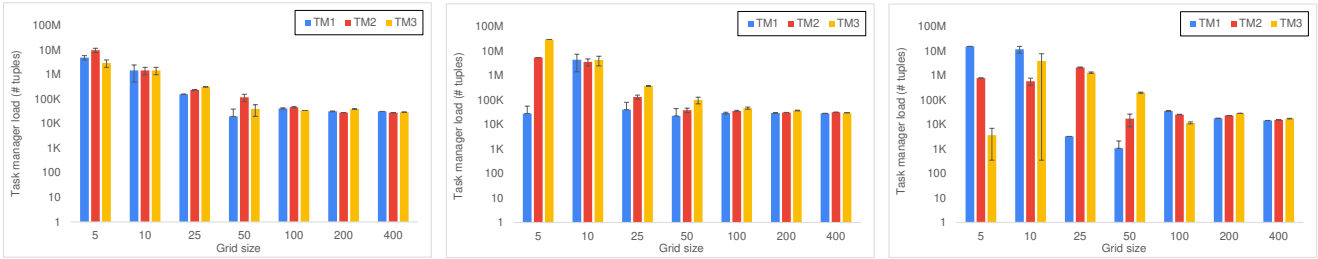
Figure 23 shows the throughput evaluation of the range

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



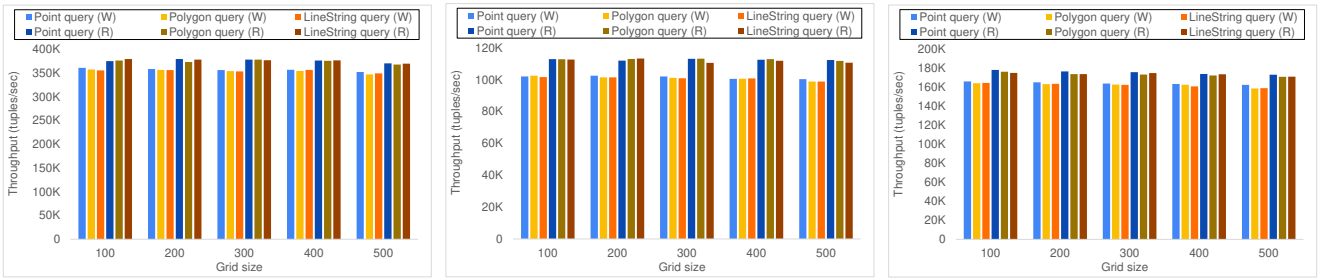
(a) Range query (b) kNN query (c) Join query

FIGURE 21. Num. of Queries: 10 (Smaller Grid size (larger cell size) results in lower grid-based pruning and imbalanced skewed data distribution across the distributed nodes; thus, resulting in lower throughput and vice versa)



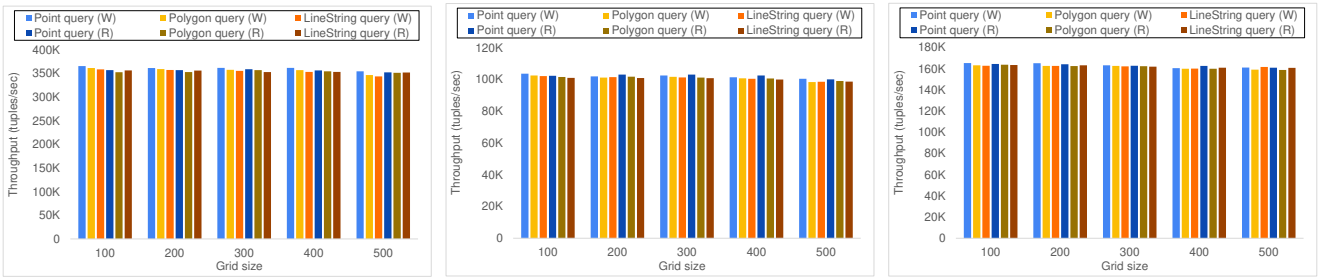
(a) Random (b) Hotspot (c) Multi-hotspot

FIGURE 22. Range query: skewed data effect on task manager load



(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 23. Range query: Varying grid size

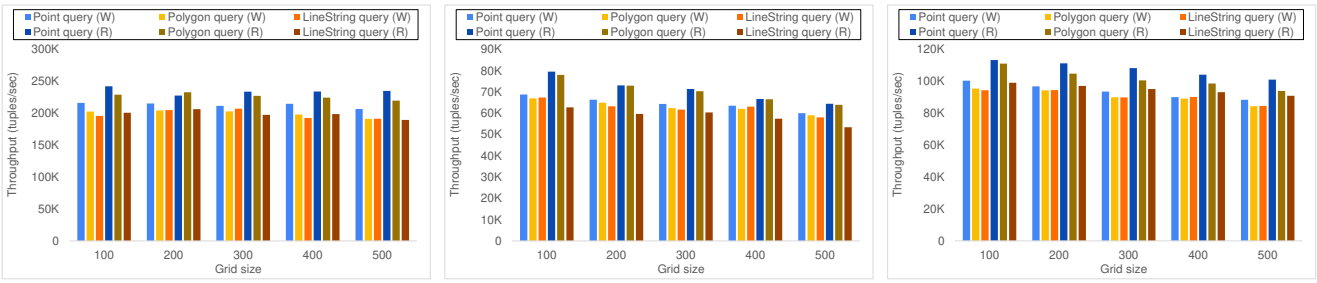


(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 24. kNN query: Varying grid size

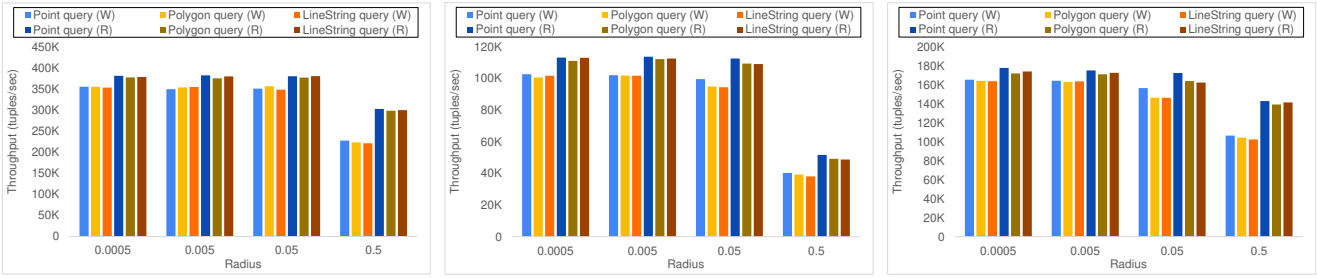
query by varying grid size (g) parameter from 100 to 500. In Figure 23(a), the query is evaluated for Point stream with Point, Polygon and LineString query objects. The throughput decreases slightly with the increase in grid size. This is due to the fact that as g increases, the number of grid cells increases quadratically. This quadratic growth in the number of cells results in increased processing (hashing) cost and thus reduced throughput. Another thing to note is that the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



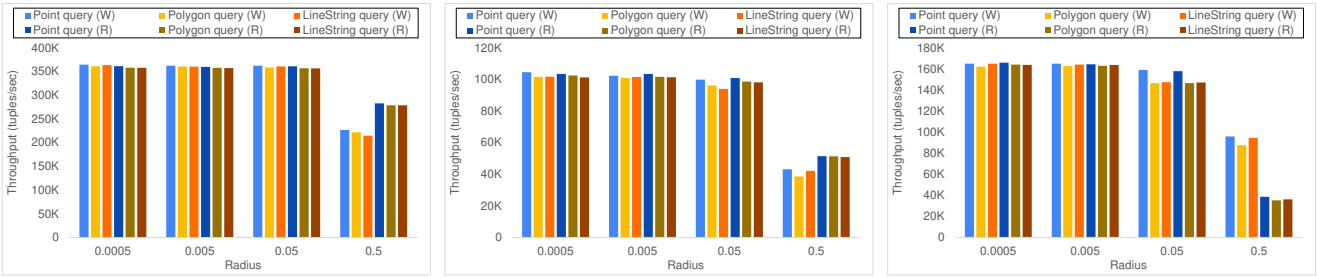
(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 25. Join query: Varying grid size



(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 26. Range query: Varying query radius



(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 27. kNN query: Varying query radius

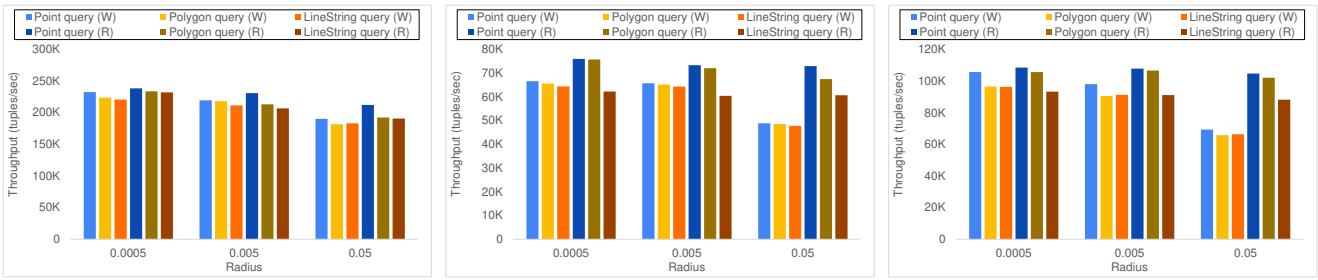
window-based range query throughput is slightly lower than its real-time counterpart. This is because the window-based query buffers stream tuples for a fixed time duration, process them as a batch and generates output corresponding to the whole window contents. In contrast, real-time range query generates output as soon as it gets an input tuple. The slightly lower throughput for the window-based query is due to the extra processing cost required in maintaining and processing window operator.

Figures 23(b) and 23(c) shows the range query evaluation for the Polygon and LineString streams, respectively. The trend in the Figures is same as that of Figure 23(a). However, the throughput for the Polygon and LineString streams is lower than the Point stream. This is obvious as the Polygon and LineString are more complex spatial objects than Point object, their distance computation is much more expensive. Thus, resulting in lower throughput.

Next, Figure 24 evaluates the kNN query by varying grid size (g) parameter. In Figure 24(a), the query is evaluated for Point stream with Point, Polygon and LineString query objects. The throughput decreases slightly with the increase in grid size due to the reason discussed above. Figures 24(b) and 24(c) show the range query evaluation for the Polygon and LineString streams, respectively. The trend in the Figures is same as that of Figure 24(a).

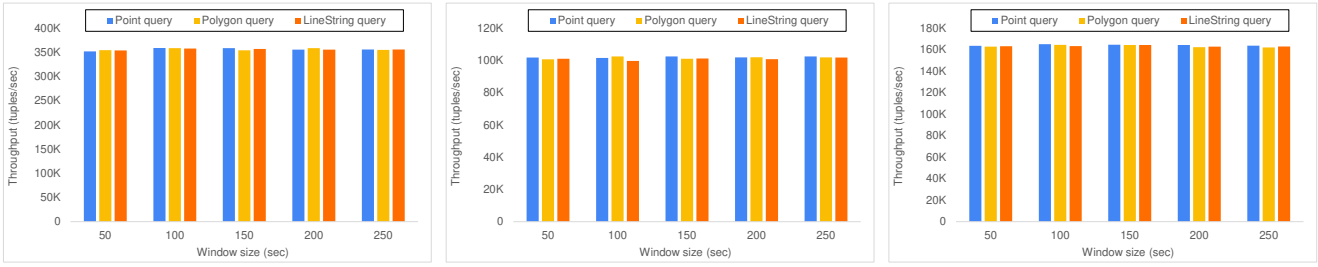
In Figure 25, join query is evaluated by varying grid size (g) parameter. In Figures 25(a), 25(b) and 25(c), the query is evaluated for the Point, Polygon and LineString streams, respectively. The evaluations follow the same trend as that of range and $k$nn queries; however, throughput of the join query is much lower. This is due to the complex nature of join query.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



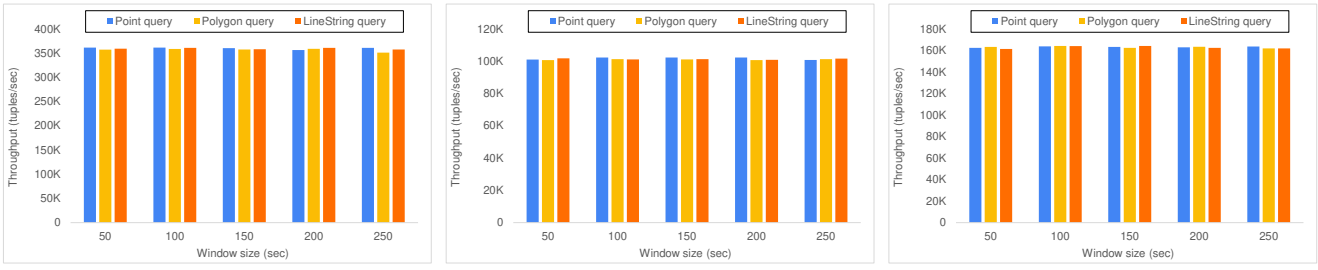
(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 28. Join query: Varying query radius



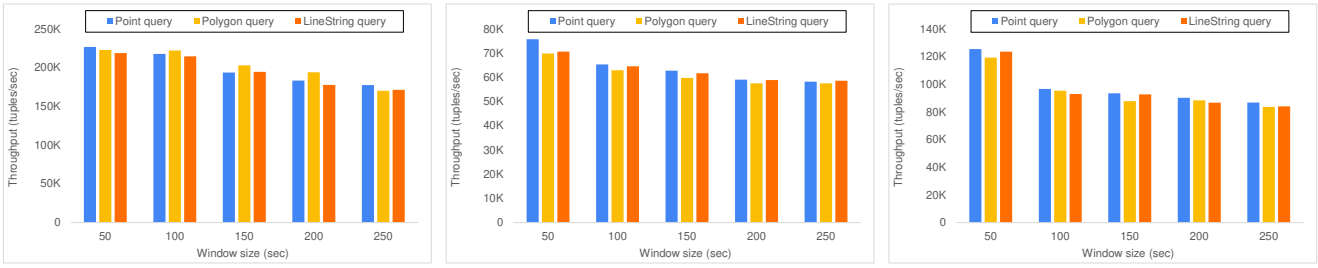
(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 29. Range query: Varying window size



(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 30. kNN query: Varying window size

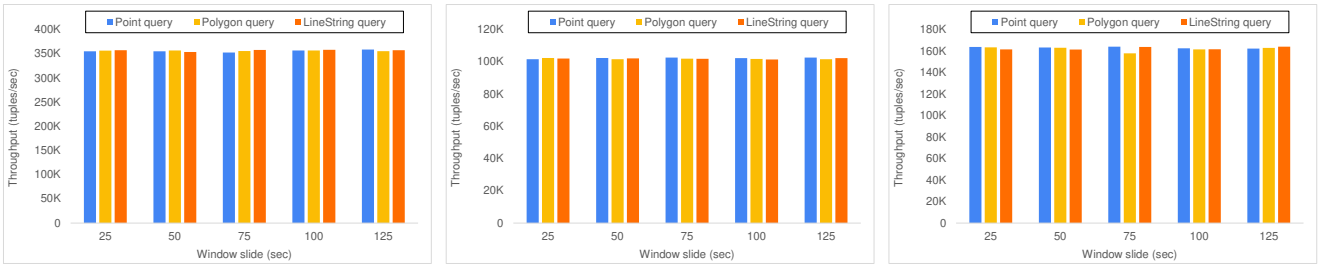


(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 31. Join query: Varying window size

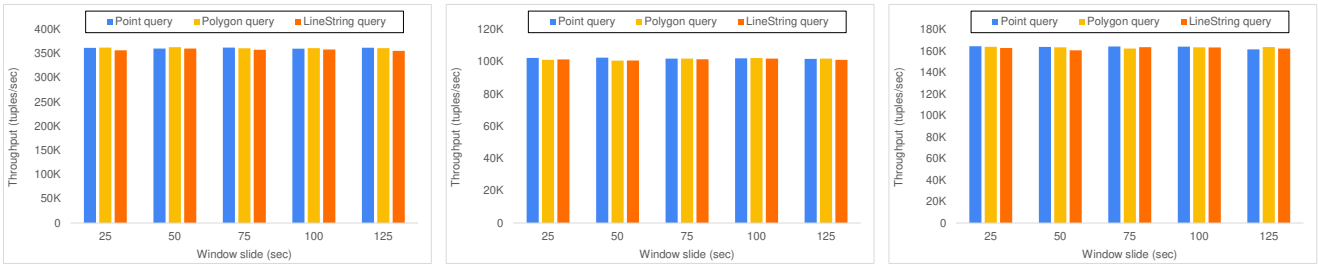
Figure 26 shows the throughput evaluation of the range query by varying query distance (r) parameter from 50 (0.0005) meters to 50,000 (0.5) meters. In Figure 26(a), the query is evaluated for Point stream with Point, Polygon and LineString query objects. From the figure, the throughput decreases with the increase in r. This is due to the fact that as r increases, pruning decreases and output increases. Thus, resulting in an increase in the computational complexity and decrease in throughput. Similar trends can be observed from Figures 26(b) and 26(c).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



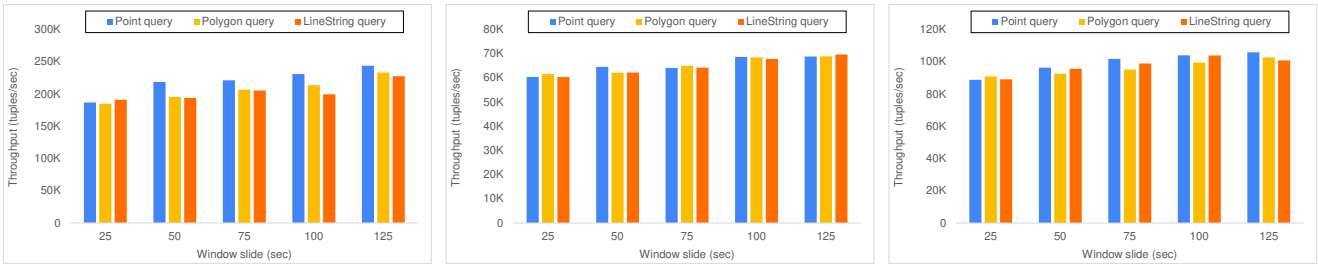
(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 32. Range query: Varying window slide size



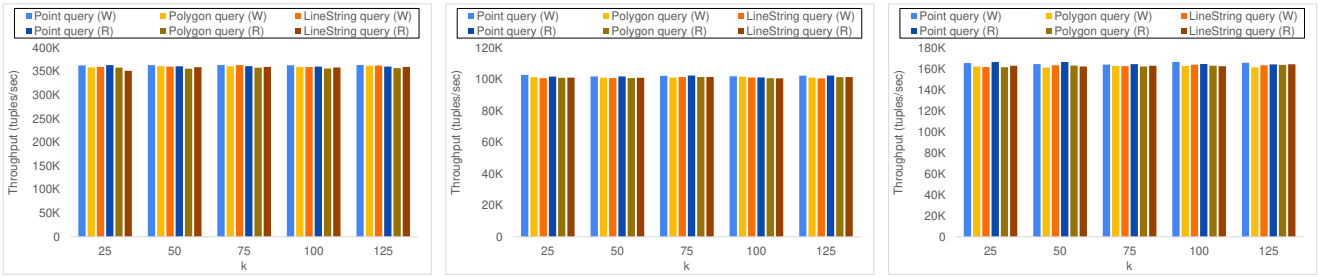
(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 33. kNN query: Varying window slide size



(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 34. Join query: Varying window slide size

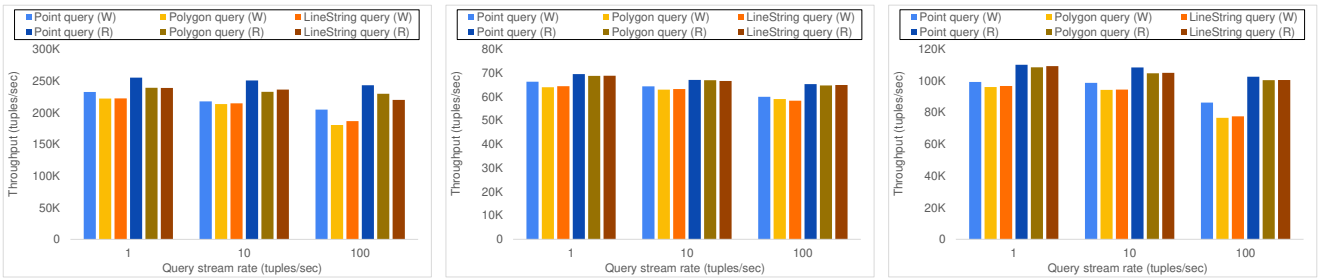


(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 35. kNN query: Varying k

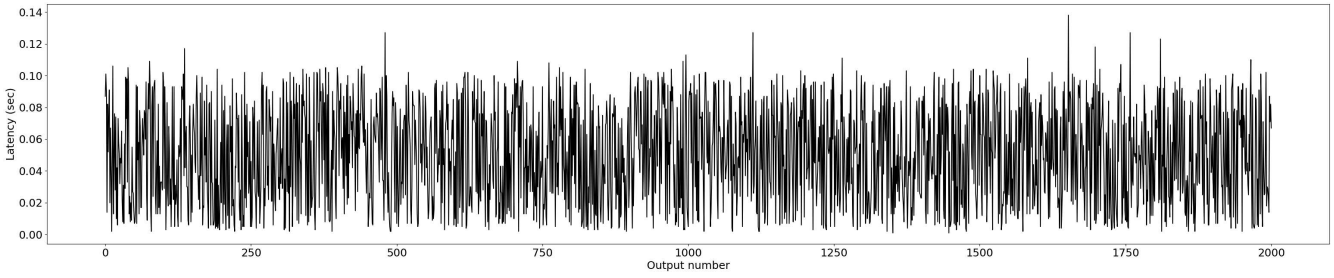
In Figure 27, kNN query is evaluated by varying query distance (r) parameter. In Figures 27(a), 27(b), and 27(c), the query is evaluated for the Point, Polygon and LineString streams, respectively. Here again the throughput decreases with increase in the query radius due to the reason discussed above. From the figures, it is very clear that we get the highest throughput for Point stream, followed by LineString and Polygon streams. This is due to the complexity of geometrical shapes. More complex the shape is, more expensive is the distance function resulting in lower throughput. In our

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access

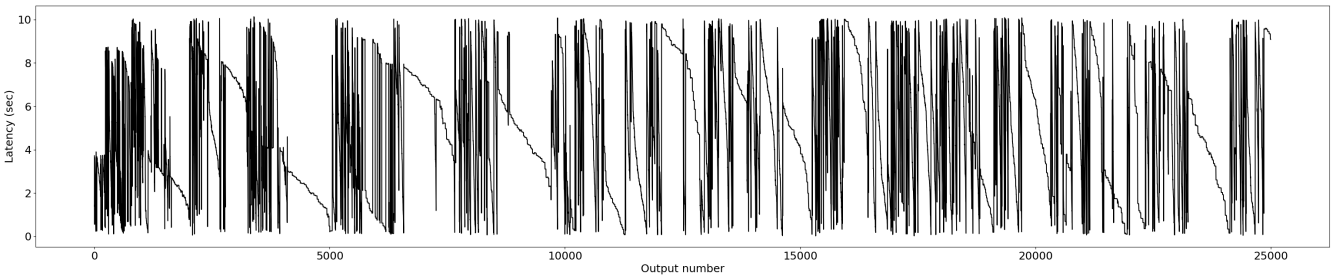


(a) Point stream (b) Polygon stream (c) Linestring stream

FIGURE 36. Join query: Varying query stream rate



(a) Realtime



(b) Window based ($W_n = 10$ sec, $W_s = 5$ sec)

FIGURE 37. Range query latency (Point stream, Polygon query)

case, Point is the simplest geometrical shape, thus, resulting in the highest throughput and so on.

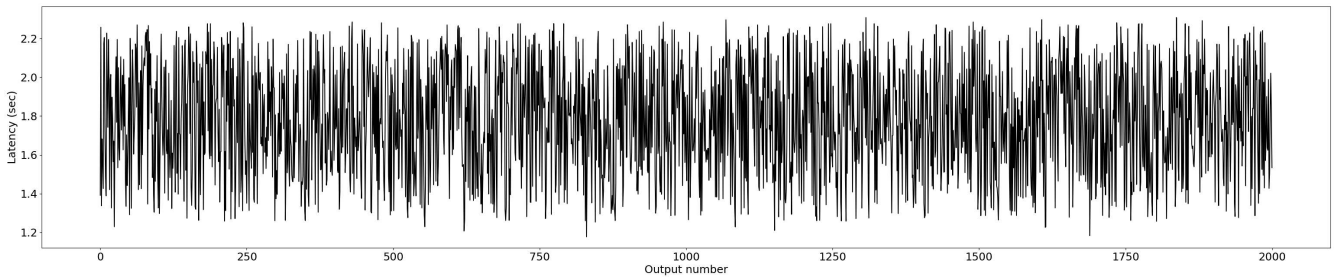
For the join query, we varied the query distance (r) parameter from 50 (0.0005) meters to 5,000 (0.05) meters as shown in Figures 28(a), 28(b) and 28(c). Here again the query follows the same trend as in the case of range and kNN query, i.e., the throughput decreases with the increase in query distance for the reason discussed earlier.

In Figure 29, we perform experiments by varying the parameter window size W_n from 50 to 250 seconds. Figures 29(a), 29(b), and 29(c) show the range query evaluation for the Point, Polygon and LineString streams, respectively. From the figures, we don't see any significant change in throughput with the increase in W_n. The reason is that the range is a computationally light query and its processing is not much affected by the window/batch size. The only difference we observe in Figures 29(a), 29(b), and 29(c) is that the throughput of the range query on Point stream is higher than the Polygon and LineString streams due to the reason discussed earlier. Similar trend can be observed for the kNN query in Figures 30(a), 30(b) and 30(c).

Figures 31(a), 31(b) and 31(c) show the join query throughput evaluation for the Point, Polygon and LineString streams, respectively, for the parameter W_n. The throughput decreases slightly with the increase in W_n. This is because, as W_n increases the number of tuples to be joined with query stream increases, resulting in a sharp increase in the number of tuples from the Cartesian product between ordinary and query steams. Thus, resulting in a slight decrease in throughput.

Next, experiments are performed by varying the parameter window slide step W_s from 25 to 125 seconds. In Figure 32, larger the parameter W_s, smaller the overlap between consecutive windows, thus, lesser processing overhead. Figures 32(a), 32(b), and 32(c) show the range query evaluation for the Point, Polygon and LineString streams, respectively. From the figures, the throughput doesn't change much with the increase in W_s because of the simple query. Similar trend

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Realtime

(b) Window based ($W_n = 10$ sec, $W_s = 5$ sec)

FIGURE 38. kNN query latency (Point stream, Polygon query)

can be observed for the kNN query in Figures 33(a), 33(b) and 33(c).

Figures 34(a), 34(b) and 34(c) show the join query evaluation for the Point, Polygon and LineString streams, respectively, for the parameter W_s. Here, in contrast to W_n, the throughput increases slightly with the increase in W_s. This is because, as W_s increases the computation overlap decreases. Precisely, if $W_s < W_n$, the window is overlapping, i.e., $W_n - W_s$ tuples overlap in the consecutive windows, Whereas, at $W_s = W_n$, the window is non-overlapping or in other words called tumbling window. Thus larger the W_s, smaller is the overlap, thus, resulting in higher throughput.

In figure 35, we performed throughput evaluation by varying parameter k from 25 to 125 for the kNN query. In Figures 35(b), 35(b) and 35(c) we varied k for Point, Polygon and LineString streams, respectively. From the figures, we don't see any change in the throughput for the different values of k. This is mainly due to the reason that kNN query is bounded by query distance r and is not much affected by varying k as long as r is constant. The throughput trend is same for all the figures, except that the throughput of Polygon and LineString streams is lower than Point stream for the reasons discussed above.

In the final throughput evaluation, Figures 36(a), 36(b) and 36(c) show the variation of query stream arrival rate from 1 to 10 for the join query. From the figures, we don't see any significant change in the throughput for the different query stream arrival rate. This is due to the strong pruning and data distribution capability of GeoFlink. Compared to GeoFlink, throughput of Apache Flink and Apache Spark Streaming deteriorates significantly with the increase in query stream arrival rate as can be observed from Figure 19.

4) GeoFlink Latency

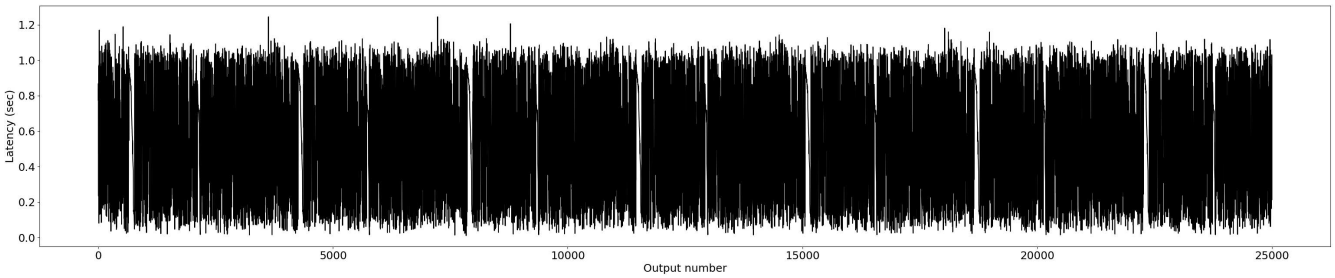
This subsection (Section IX-B4) evaluates the latency of the real-time and window-based queries. In particular, we evaluated real-time and window-based range, kNN and join queries in Figures 37, 38 and 39, respectively. The latency is evaluated in seconds (sec). For the latency evaluation, W_n = 10 seconds, W_s = 5 seconds and ω = 1 second is used.

Figures 37(a) and 37(b) show the latency graphs of real-time and window-based range queries. The latency of real-time range query is low and does not change much as GeoFlink processes each incoming tuple as it arrives. Whereas, the latency of window-based range query varies between 0 to 10 seconds because of W_n = 10 seconds. Thus, the tuple which arrives at the very start of the window has a latency of size W_n, i.e., 10 seconds because it has to wait for W_n duration to be processed. The latency decreases for the tuples arriving later in the window as can be observed from Figure 37(b).

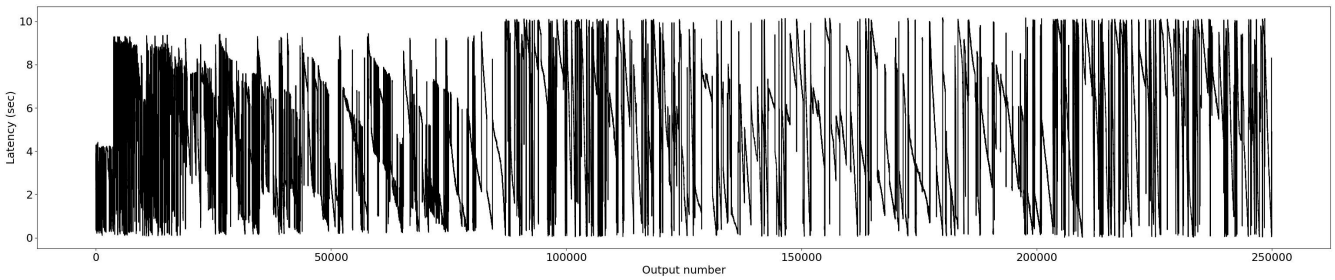
In Figures 38(a) and 38(b) latency of real-time and window-based kNN queries are evaluated. The latency of realtime kNN query follows the trend of realtime range query. However, the latency is much higher given the kNN query complexity, i.e., it takes much time to process a tuple by kNN query compared to the range query. Furthermore, realtime kNN query utilizes recency parameter ω for the kNN computation, adding ω duration to latency. The latency of the window-based varies between 0 to 11 seconds as can be observed from Figure 38(b) due to the reason discussed above.

Figures 39(a) and 39(b) show the latency graphs of real-time and window-based join queries. Given ω = 1 second, the latency of real-time join query varies mainly between

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3154063, IEEE Access



(a) Realtime



(b) Window based ($W_n = 10$ sec, $W_s = 5$ sec)

FIGURE 39. Join query latency (Point stream, Polygon query stream)

0 and 1 second. The latency of window-based join query follows the trend of window-based range and join queries, i.e., the latency varies between 0 to 10 seconds due to W_n = 10 seconds.

X. CONCLUSION AND FUTURE WORK

This work presents GeoFlink which extends Apache Flink to support spatial data types, index and continuous queries. To enable efficient processing of continuous spatial queries and for the effective data distribution among the Flink cluster nodes, a gird-based index is introduced. The grid index enables the pruning of the spatial objects which cannot be part of a spatial query result and thus guarantees efficient query processing. Furthermore, it helps in uniform data distribution across distributed cluster nodes. GeoFlink supports spatial range, spatial kNN and spatial join queries on Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon spatial objects. Besides, GeoFlink supports incoming data streams in GeoJSON, CSV and WKT formats. Extensive experimental study proves that GeoFlink results in higher throughput than Apache Flink and Apache Spark Streaming. Furthermore, experiments prove that grid index performs good for uniform and skewed data distributions if the grid size is chosen carefully. As a future direction, we are working on GeoFlink's extension to support complex spatial query operators involving join between stream and static data. Furthermore, we are looking into other efficient spatial index structures for spatial stream processing.

XI. ACKNOWLEDGMENT

This research was partly supported by JSPS KAKENHI Grant Numbers JP20K19806 and JP19H04114, and projects JPNP18010 and JPNP20006, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] PostGIS, "PostGIS: Spatial and Geographic objects for PostgreSQL," http://postgis.net/, [Online; accessed 10-March-2020].

[2] QGIS, "Qgis, a free and open source geographic information system," https://qgis.org/en/site/, 2020, [Online; accessed 31-March-2020].

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–438. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[4] ApacheFlinkDoc, "Dataflow Programming Model," https://ci.apache.org/projects/flink/flink-docs-stable/concepts/programming-model.html, 2019, [Online; accessed 06-November-2019].

[5] T. A. S. Foundation, "Apache Samza - Distributed Stream Processing," http://samza.apache.org/, [Online; accessed 11-November-2018].

[6] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: A high performance spatial data warehousing system over mapreduce," *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1009–1020, Aug. 2013.

[7] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *2015 IEEE 31st ICDE*, 2015, pp. 1352–1363.

[8] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: the geospark perspective," *GeoInformatica*, vol. 23, no. 1, pp. 37–78, 2019.

[9] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys, "Trees or grids?: indexing moving objects in main memory," in *17th ACM SIGSPATIAL, Proceedings*, 2009, pp. 236–245.

[10] S. A. Shaikh, K. Mariam, H. Kitagawa, and K.-S. Kim, "Geoflink: A distributed and scalable framework for the real-time processing of spatial streams," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*, ser. CIKM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3149–3156.

[11] ESRI, "Esri: See patterns, connections, and relationships," https://www.esri.com/, [Online; accessed 12-November-2019].

[12] J. Lu and R. Güting, "Parallel secondo: Boosting database engines with hadoop," 12 2012, pp. 738–743.

[13] J. N. Hughes, A. Annex, and et al., "GeoMesa: a distributed architecture for spatio-temporal fusion," in *Geospatial Informatics, Fusion, and Motion Video Analytics V*, vol. 9473, 2015.

[14] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *2015 31st IEEE ICDE Workshops*, 2015, pp. 34–41.

[15] M. Tang, Y. Yu, W. G. Aref, A. R. Mahmood, Q. M. Malluhi, and M. Ouzzani, "Locationspark: In-memory distributed spatial query processing and optimization," *ArXiv*, vol. abs/1907.03736, 2019.

[16] F. Baig, H. Vo, T. M. Kurç, J. H. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *Proceedings of the 25th ACM SIGSPATIAL*. ACM, 2017, pp. 28:1–28:10.

[17] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1507–1518.

[18] A. Storm, "Apache Storm: Distributed realtime computation system," https://storm.apache.org/, [Online; accessed 10-March-2020].

[19] F. A. Ahmed, J. Ye, and J. Arthur, "Evaluating streaming frameworks for large-scale event streaming," https://medium.com/adobetech/evaluating-streaming-frameworks-for-large-scale-event-streaming-7209938373c8, 2019, [Online; accessed 10-March-2020].

[20] F. Zhang, Y. Zheng, D. Xu, Z. Du, Y. Wang, R. Liu, and X. Ye, "Real-time spatial queries for moving objects using storm topology," *ISPRS International Journal of Geo-Information*, vol. 5, no. 10, p. 178, Sep 2016.

[21] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. M. Basalamah, "Tornado: A distributed spatio-textual stream processing system," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 2020–2023, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p2020-mahmood.pdf

[22] A. R. Mahmood, A. Daghistani, A. M. Aly, M. Tang, S. M. Basalamah, S. Prabhakar, and W. G. Aref, "Adaptive processing of spatial-keyword data over a distributed streaming cluster," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, F. B. Kashani, E. G. Hoel, R. H. Güting, R. Tamassia, and L. Xiong, Eds. ACM, 2018, pp. 219–228. [Online]. Available: https://doi.org/10.1145/3274895.3274932

[23] Y. Chen, Z. Chen, G. Cong, A. R. Mahmood, and W. G. Aref, "SSTD: A distributed system on streaming spatio-textual data," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2284–2296, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2284-chen.pdf

[24] R. Metzger, "Scaling flink automatically with reactive mode," https://flink.apache.org/2021/05/06/reactive-mode.html, [Online; accessed 10-May-2021].

[25] T. A. S. Foundation, "Apache Kafka - A Distributed Streaming Platform," http://spark.apache.org/, [Online; accessed 11-November-2018].

[26] V. Robert Metzger, "Kafka + flink: A practical, how-to guide," https://www.ververica.com/blog/kafka-flink-a-practical-how-to, 2020, [Online; accessed 14-December-2020].

[27] P. Brebner, "The power of kafka partitions : How to get the most out of your kafka cluster," https://www.instaclustr.com/the-power-of-kafka-partitions-how-to-get-the-most-out-of-your-kafka-cluster/, 2020, [Online; accessed 14-December-2020].

[28] J. T. Henrikson, "Completeness and total boundedness of the hausdorff metric," in *MIT Undergraduate Journal of Mathematics*, 1999, pp. 160–166.

[29] A. Dumitrescu and G. Rote, "On the fréchet distance of a set of curves." in *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04), Montreal, August 9-11*, 01 2004, pp. 162–165.

[30] A. Efrat, L. Guibas, S. Har-Peled, J. Mitchell, and Murali, "New similarity measures between polylines with applications to morphing and polygon sweeping," *Discrete and Computational Geometry*, vol. 28, pp. 535–569, 07 2002.

[31] W. Rucklidge, Ed., *The Hausdorff distance*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 27–42. [Online]. Available: https://doi.org/10.1007/BFb0015093

[32] M. Davis, "Jts topology suite: a java api for geometric operations," https://sourceforge.net/projects/jts-topo-suite/, 2021, [Online; accessed 20-April-2021].

[33] M. Hadjieleftheriou, Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, *R-Trees: A Dynamic Index Structure for Spatial Searching*. Cham: Springer International Publishing, 2017, pp. 1805–1817.

[34] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, *Spatial Indexing Techniques*. Boston, MA: Springer US, 2009, pp. 2702–2707.

[35] R. H. Guting, "An introduction to spatial database systems," *VLDB Journal*, vol. 3, pp. 357 – 399, 1994.

[36] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 397–409, Dec. 1979.

[37] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in *Proceedings of the 17th ACM SIGKDD*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 316–324.

[38] D. Brscic, T. Kanda, T. Ikeda, and T. Miyashita, "Person tracking in large public spaces using 3-d range sensors," *Human-Machine Systems, IEEE Transactions on*, vol. 43, pp. 522–534, 11 2013.

[39] N. B. Footprints, "Shapefile of footprint outlines of buildings in new york city," https://data.cityofnewyork.us/Housing-Development/Building-Footprints/nqwf-w8eh, [Online; accessed 05-August-2021].

[40] S. A. Shaikh, "Kafka timeseries producer," https://github.com/salmanahmedshaikh/KafkaTimeSeriesProducer, 2021, [Online; accessed 05-August-2021].

[41] N. I. of Advanced Industrial Science and T. (AIST), "Aist artificial intelligence cloud (aaic)," https://www.airc.aist.go.jp.