EasyChair Preprint

№ 1548

A Distance-Window Based Real-Time Processing of Spatial Data Streams

Salman Shaikh, Akiyoshi Matono and Kyoung-Sook Kim

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

September 22, 2019

A Distance-window based Real-Time Processing of Spatial Data Streams

Salman Ahmed Shaikh, Akiyoshi Matono, Kyoung-Sook Kim
*Artificial Intelligence Research Center
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan
{shaikh.salman,a.matono,ks.kim}@aist.go.jp

***Abstract*—Real-time and continuous processing of citywide spatial data is an essential requirement of smart cities to guarantee the delivery of basic life necessities to its residents and to maintain law and order in it. With the availability of low cost 3D scanners recently, citywide 3D spatial data can be obtained easily. 3D spatial data contains a wealth of information including images, point-cloud, GPS/IMU measurements, etc., and can be of potential use if integrated, processed and analyzed in real-time. The 3D spatial data is generated as continuous data stream, however traditionally it is processed offline. Many smart city applications require real-time integration, processing and analysis of spatial stream, for-instance, forest fire management, real-time road traffic analysis, disaster engulfed areas monitoring, people flow analysis, etc., however they suffer from slow offline processing of traditional systems. To make the most of this wealthy data resource, it must be processed and analyzed in real-time. This paper presents a framework for the continuous and real-time processing and analysis of 3D spatial streams. Furthermore, we propose a distance-based window for the continuous queries over 3D spatial streams. An experimental evaluation is also presented to prove the effectiveness of the proposed framework and the distance-based window.**

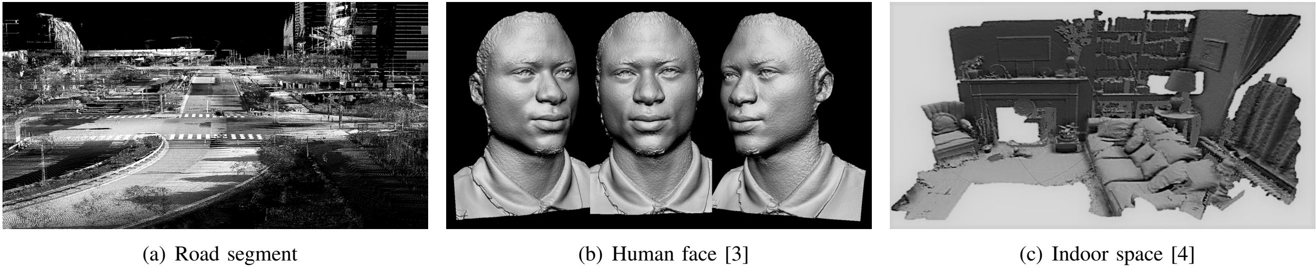
***Index Terms*—3D spatial data, Data stream, Point-cloud data, Real-time processing, Continuous query, Distance-based window**

I. Introduction

Smart cities development and management require real-time processing and analysis of citywide data. With the increase in the availability of inexpensive 3D scanners these days, detailed citywide 3D scanning data of earth surfaces (planes, trees), road segments, building interior and exterior can be obtained easily. 3D spatial data is usually generated as a stream of points by mounting scanning equipment on drones and cars for large area scanning or on some stationary platform for small area scanning or indoor scanning. Fig. 1 shows different types of 3D scanning, for instance, a road segment (Fig. 1(a)), a human face (Fig. 1(b)) and an indoor space (Fig. 1(c)). 3D spatial data streams or spatial mapping data obtained through 3D scanners provides a detailed representation of real-world surfaces. 3D spatial data streams, in contrast to 2D spatial data streams which consist of geographical location in terms of longitude and latitude, consist of images, GPS/IMU measurements and point-cloud [1]. Images and GPS/IMU measurements are obtained by mounting cameras and GPS/IMU devices on 3D data collection platform respectively, whereas the point-cloud data is obtained through optical scanners (e.g., LiDAR - light detection and ranging) that uses laser light to densely sample the earth's surface, producing highly accurate x,y,z measurements. In addition, each point in the point-cloud may contain some or all of the following attributes: intensity, return number, number of returns, RGB (red, green, and blue) values, GPS time, edge of flight line, scan angle and scan direction [2].

3D scanning data is generated continuously as a stream of 3D points and images, as the platform on which the scanners are mounted moves. The 3D spatial stream is voluminous. For instance, aerial scanning of a road segment (Fig. 1(a)) consists of 5-10 points per square meter resulting in a set of billions of points for an average city [5]. Similarly, 3D stream is velocious, for instance an MMS (Mobile Mapping System) vehicle can move at a speed of 80 kilometers/hour resulting in the generation of millions of points per second [6]. 3D spatial data is generated as a stream however, traditionally it is processed offline due to the inherent noise, huge volume and high velocity, i.e., the data is first stored on a secondary storage, denoised and is then queried and/or analyzed. Examples include [5], [7], [8], where the authors assumed static point-cloud available in a HDFS cluster or some distributed database and made use of distributed computing for its processing [9]. However, many applications require real-time integration, processing and analysis of 3D spatial streams. For instance: 1) The spatial stream from several airborne lidar sensors must be integrated in real-time and analyzed interactively to manage forest fire, 2) The spatial stream of road traffic must be monitored in an online fashion to manage road traffic and identify speeding vehicles, 3) Citywide people flow must be analyzed in real-time to better manage the city resources, etc. These and other similar applications suffer from the traditional offline processing which cannot handle continuously arriving spatial stream in real-time. Integrating, processing and analyzing 3D spatial stream in real-time is challenging due to its size, structure and inherent noise.

Keeping in view the importance of real-time processing and analysis of 3D spatial stream, this work presents and demonstrates a robust distributed framework named "CQ3D". The proposed framework takes raw 3D spatial streams as input, pre-processes/cleans them, performs different point-cloud operations (object detection, segmentation, etc.) on them, executes different continuous queries on the resultant



(a) Road segment (b) Human face [3] (c) Indoor space [4]

Fig. 1. Different types of 3D scanning

data (filtration, projection, join, etc.) and produces real-time continuous output which may be used for analysis. Since the streams are unbounded, a bounding mechanism is needed to execute different types of queries on it. Traditionally count-based or time-based windows are used to limit the amount of data for query processing over data streams. However such windows do not make much sense on spatial data stream as the queries are mostly related to the trajectories of moving platform or moving objects. Therefore we require a mechanism to bound the length of moving object trajectories. Hence, we propose and define a distance-based window to execute continuous queries on spatial data streams. This work assumes that the 3D spatial streams are ordered.

The main challenges in the development of CQ3D include the handling of noisy, high volume and high speed data in real time, which require distributed and scalable computing resources. Furthermore, due to the absence of standardized 3D spatial data format, the incoming data from different scanning devices consist of varying number of formats and different number of attributes which must be pre-processed to generate a uniform input for CQ3D, which is quite challenging. 3D spatial data operations are computationally expensive, hence their real time processing require distributed deployment, which is quite challenging. Furthermore, to identify if a certain part of a trajectory lies within the boundaries of distance-based window, distance between consecutive spatial points need to be computed. Considering a stream of moving objects, a large number of distance computations are required in real-time which is another challenging part of this work.

In the proposed framework we address the above challenges with the following contributions:

- A continuous querying framework for the real-time processing of spatial data streams
- Semantics of a distance-based window
- Experimental evaluation to prove the effectiveness of the proposed framework and the distance-based window

The rest of the paper is organized as follows. Section II discusses the related work. Section III discusses general continuous query processing over data streams and windowing. Section IV presents the proposed distance-based window. In Section V, the proposed continuous querying framework for 3D spatial stream is presented. Detailed experimental evaluation is presented in Section VI while Section VII concludes this paper and discusses some of the future directions.

II. Related Work

Recent past era has witnessed development of many general Stream Processing Engines (SPEs). Apache Spark Streaming [10], Stanford STREAM [11], Apache Samza [12] are among the state-of-the-art SPEs and are being used by many data giants including Amazon, TripAdvisor, Yahoo, etc. to process their business data streams. Current era is witnessing a new type of big data, i.e., 3D spatial data or point-cloud data with several applications, however existing SPEs do not provide functionalities/operators to process it efficiently. Typical processing over point-cloud data includes object identification, segmentation, difference detection (between two point-cloud versions), Digital Elevation Model (DEM) construction [8], etc., which are not supported by any of the existing stream processing engine. Most of the existing point-cloud data processing frameworks [5], [7] can handle only static point-cloud and are too slow to support its real-time processing and analysis.

Beside the above mentioned SPEs, there exist a few continuous 3D spatial stream processing solutions. For instance, [13] gave a Euclidean-distance based approach for the continuous clustering of point-cloud data. In their work, given a set of point-cloud the proposed algorithm identifies a disjoint group of point that can be potential objects surrounding the sensor. The proposed algorithm uses single pass to cluster the LiDAR point-cloud data. To achieve the continuous processing, the authors make use of fine-grained pipeline. Authors in [14] proposed an algorithm for segmenting a continuous stream of 3D range data in real-time. The proposed algorithm computes normal vectors of incoming points from their local neighborhood and clusters the new points by assessing their Euclidean and angular distance to previously clustered points. Kim et al. gave a continuous query approach for 3D objects [15]. However their focus was perspective query. Perspective query requests data with different levels of detail according to the importance of data. In contrast, this work presents a distributed framework for the real-time processing and analysis of 3D spatial data stream.

Since a data stream is an infinite sequence of tuples, windowing spatial data streams is an important research issue. The works [16], [17] proposed methods for approximate join computation over data streams by using sliding windows. GrubJoin [18] considers sliding-window join with CPU load shedding. Grubjoin uses window-harvesting which picks the

most profitable segments of individual windows for the join processing, in an effort to maximize the join output rate. However, the existing windows are either count-based or time-based, which cannot answer continuous queries by bounding the length of the trajectories. Hence, we propose a distance-based window to execute continuous queries on spatial data streams.

III. Continuous Query Processing and Windowing

This section presents a quick overview of general continuous query processing and the use of windows for it. Data stream is an unbounded, or at least unknown, collection of events which arrive continuously and usually at high velocity. In order to process and query continuously evolving data streams, many Stream Processing Engines (SPEs) have been developed. Spark [19], Samza [12], Kafka [20], etc. are some of the well-known and commonly used SPEs. When a user registers a Continuous Query (CQ) to an SPE, it is executed continuously on the newly arriving stream tuples and generates continuous output. Usually the CQ output is generated incrementally, i.e., the output is generated only for the stream tuples which arrived after the generation of last output.

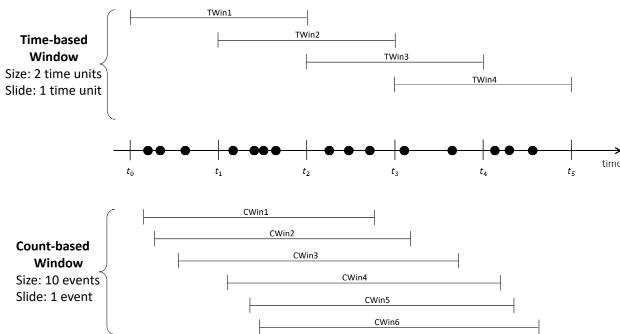


Fig. 2. A comparison of time-based and count-based windows

Since the data stream is unbounded, *windowing* is usually used for the execution of stateful operators (relational join, aggregation, etc.). Windowing splits the unbounded data stream into finite sets, or windows, based on some specified criteria. A window is an in-memory table in which events/tuples are added and removed based on a set of policies. Golab [21] suggested two models for windows: 1) Fixed (tumbling window): The fixed window model periodically clears the accumulated data. That is, a stream is divided into non-overlapping partitions and data are kept only for that part of a stream which falls within the current partition. 2) Sliding: The sliding window model expires old events as new events arrive. That is, a stream is divided into overlapping partitions of newest events. Both the window models can be broadly divided into two types: 1) Time-based: Stores only those events which have been generated or have arrived in the last Σ time units. E.g., Events/Tuples generated in last 24 hours. 2) Count-based: Stores the N newest events. E.g., A window consisting of recent 1 million tuples.

Fig. 2 shows the time-based (above time line) and count-based (below time line) sliding windows. Assume a time ordered stream of events between t_0 and t_5. The size of time-based window is 2 time units with a slide step of 1 time unit, i.e., the window stores events which have been generated in the last 2 time units and slides with the step of 1 time unit. In Fig. 2, the boundaries of time-based window 1 (TWin1) is from t_0 to t_2, which slides with the availability of new data at t_3 and causes the generation of new window (TWin2) with boundaries t_1 to t_3. On the other hand, the count-based window in Fig. 2 contains 10 events. Since the slide step in case of count-based window is 1 event, the window slides with the arrival of each new event to generate new window, i.e., CWin1, CWin2, CWin3,..., as can be observed from Fig. 2.

IV. Distance-based Window

Depending upon the mounting of 3D scanner i.e., on static platform (case I) or on mobile platform (case II), two different types of spatial data stream can be generated. In case I, the data stream is generated for a fixed region r for the course of time. Whereas in case II, the spatial stream is generated for the changing surroundings as the platform moves. Both the cases have different applications and can answer different queries. For instance in case I, a user may be interested in difference detection, i.e, identifying the change in surroundings during the course of time that may include counting the number of vehicles passing the region r over the course of time, computing the crowd flow, or detecting the changes in the surroundings' structure with respect to time. Such queries can be answered via our proposed framework CQ3D (Sec. V) by utilizing time-based window discussed in Sec. III. For the case II (i.e., mobile platform), the queries are mainly based on trajectories or length of trajectories.

Example 1: Assume a mobile platform m scanning a road segment through 3D scanner and generating a stream of 3D spatial data. As the data stream arrives, the user is interested in detecting the different objects (people, cars, trees, etc.) in it and execute continuous queries on it by bounding the stream for a particular length of trajectory. Let the user would like to know the number of different objects detected in last x distance units. Since the speed of m is not uniform, using the existing window semantics, one can obtain the number of objects detected in last t time units however cannot obtain the number of objects detected in last x distance units. We call the window, capable of bounding or tracking the length of trajectory to answer the continuous queries, the *distance-based window*.

Distance-based window can be useful for many real-world spatial continuous queries and perspective queries, however it is not supported by any of the existing SPE or geo-spatial platform. Keeping in view its importance, this work proposes the distance-based window. In the following, we formally define the distance-based window semantics and the distance functions.

A. Window Semantics

Let Γ be a discrete time domain with total order $\leq$. A time instant t is any value from Γ, which denotes the application time, i.e., the time of the event occurrence and not the system time. For the sake of simplicity, we assume that the Γ is a set of non-negative integers $\{0, 1, 2, 3, ...\}$. Let S is a potentially infinite ordered sequence of events e, obtained from a moving object. Each $e \in S$ consists of an object's unique id (ι), its spatial location λ in terms of longitude and latitude at timestamp $t \in \Gamma$ and a timestamp $t \in \Gamma$, forming a spatial tuple of the form: (ι, λ, t).

- **Trajectory:** Consider a moving object o_i, generating a continuous stream of events. By sampling the movement of o_i, a polyline p is obtained. In geometrical terms, this polyline is called trajectory [22]. Fig. 3 shows the movement of a spatial object (spatial trajectory) in 2 dimensional space and its corresponding spatio-temporal trajectory in 3 dimensional space. As shown in the figure, a trajectory can be represented by a polyline.
- **Trajectory length:** It can be defined as the length of the polyline, which is approximately equal to the ground distance covered by the spatial moving object (spatial trajectory). Consider a spatial trajectory polyline p_i (similar to the one shown in Fig. 3), composed of the ordered events, $e_{i1}, e_{i2}, ..., e_{ij}$, where $e_{i1}.t < e_{i2}.t < ... < e_{ij}.t$, then the trajectory length corresponding to the p_i is given by:

$$TLength(e_{i1}, e_{ij}) = \sum_{a=1}^{j-1} dist(e_{ia}.\lambda, e_{i(a+1)}.\lambda) \quad (1)$$

  where the function $dist$ denotes the geographical distance between two event locations, discussed in Sec. IV-B.

We can now define the distance-based window as follows:

Definition 1: **Distance-based Window:** Assuming a set of n moving objects $O = \{o_1, o_2, ..., o_n\}$ with trajectory polylines $P = \{p_1, p_2, ..., p_n\}$. A distance-based window of size x contains the events' tuples corresponding to the trajectories of moving objects in O, such that the trajectory length of each $o_i \in O$ is at most x.

According to the Definition 1, the distance window contains the latest trajectory tuples such that the trajectory length is at most the window size x. Older trajectory tuples must be deleted periodically to keep the trajectory length within x. To achieve this, for each new trajectory event tuple that is inserted into the distance-window, the length of its corresponding trajectory is computed. If the length is greater than x, the oldest trajectory tuples are deleted recursively until the trajectory length is at most x. To support efficient insertion and deletion of the new and the old trajectory tuples, doubly linked list data structure is used for the distance-based window with $O(1)$ insertion and deletion complexities.

A distance-based window can be tumbling (fixed) or sliding. In a tumbling window, spatial stream is divided into non-overlapping partitions and the stream tuples are kept only for that part of the stream which falls within the current partition, whereas in the distance-based sliding window, the old tuples expire as new tuples arrive and is divided into overlapping partitions of newest tuples.

B. Distance Functions

This section discusses the available geographic distance functions to compute the trajectory length. Although there exist a number of geographic distance functions, in this work we make use of the following three: 1) Haversine, 2) Equirectangular, and 3) Spherical law of cosines.

- **Haversine:** The Haversine formula calculates the great-circle distance between two points, i.e., the shortest distance over the earth's surface, giving an "as-the-crow-flies" distance between the points (ignoring any hills they fly over). The word haversine comes from the function: $haversine(\theta) = Sin^2(\theta/2)$. For the details on how the haversine function can be used to compute the distance between two geographical coordinates, please refer [23].
- **Equirectangular approximation:** As the name indicates, equirectangular is an approximate geographic distance function. This function uses only one trigonometric function compared to the seven trigonometric functions used by the haversine function, to reduce the computation cost at a small cost of accuracy. For details, please refer [23].
- **Spherical law of cosines:** This distance function is as accurate as the haversine function, however makes use of six trigonometric functions compared to the seven trigonometric functions used by the haversine function. Spherical law of cosines is slightly less computationally expensive than the haversine. For details, refer [23].

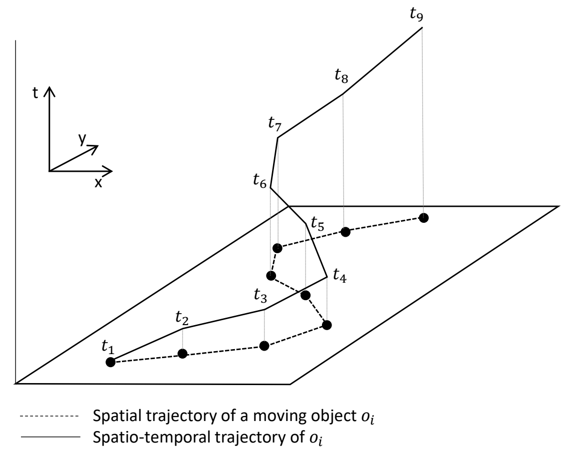


Fig. 3. The movement of spatial object and the corresponding trajectory

V. A Framework for the Real-time Processing of 3D Spatial Streams

This section presents a framework for the real-time processing of 3D spatial streams. To facilitate continuous queries

(CQs) capable of bounding spatial trajectory length, the proposed framework implements the distance-based window proposed in Sec. IV. In the following, we present the architecture and the query processing of the proposed framework.

A. Architecture

The architecture of the proposed framework is shown in Fig. 4. Its main components include *pre-processor*, *existing point-cloud operators*, *continuous query processor* and a *persistent storage*. The 3D spatial data format varies drastically depending upon the 3D scanning technology and the combination of devices used. 3D spatial data may also contain missing values and noise. The purpose of the *pre-processor* is to provide integrated, clean and uniform data to all the components of the framework. For instance, an MMS vehicle generates synchronized streams of images, GPS/IMU measurements and point-cloud [1]. Different spatial operators require different streams in different formats, for instance some object detection algorithms require both the image and point-cloud streams, while others require only image stream or only point-cloud stream, in addition to the GPS/IMU measurements. Hence, depending upon the requirements, different streams are integrated and cleaned and are available to the *existing point-cloud operators* and *continuous query processor* components of our framework.

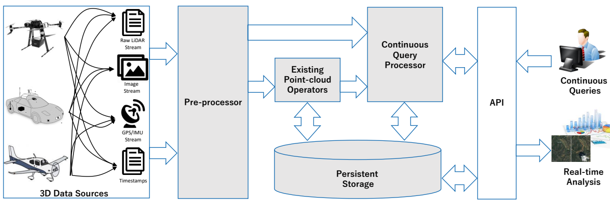


Fig. 4. Framework for the real-time processing of spatial streams

There exist a number of libraries providing real-time operations over point-cloud data including segmentation [14], object detection [24], etc. The *existing point-cloud operators* enables the integration of the existing point-cloud libraries to our framework rather than reinventing the wheel for all the point-cloud operations. The *continuous query processor* on the other hand can either take input from the *pre-processor* or the *existing point-cloud operators*. This component is meant to support continuous queries directly over 3D spatial data streams and/or on the output of the *existing point-cloud operators* component. The *persistent storage* stores the output of different operations over 3D spatial data for later use. Furthermore, various operations involve join between fresh stream and historical data, e.g., change detection algorithm, where the historical data may be obtained from the *persistent storage* component.

For the development of highly distributed, scalable and fault-tolerant pipeline corresponding to the proposed CQ3D architecture, we propose the use of Apache Spark Streaming and Apache Kafka. Apache Spark [10] is a distributed in-memory framework for handling batch, real-time analytics, and data processing workloads, while Apache Kafka [20] is a distributed streaming platform for building real-time data pipelines and stream processing applications. Kafka uses *topics* to store data tuples. The reasons for using spark streaming are multifold. Compared to other state-of-the-art streaming engines, it supports many APIs including Scala, Java and Python. Furthermore it is the main memory based system, i.e., there is no I/O overhead, hence resulting in throughput of millions of tuples per second on a single computing node. Similarly Apache Kafka is a scalable, high performance, low latency messaging system, highly supported and recommended by Spark Streaming for building high throughput production pipelines. In our framework, Spark Streaming is used as a distributed CQ processor whereas Kafka is used as a distributed messaging system to provide communication between the different components of the framework. For the persistent storage, we made use of Apache Cassandra, which is a distributed, fault tolerant and highly scalable database. Since our main requirement from persistent storage is to speed-up data retrieval for real-time data/join processing, Cassandra supports it with the use of columnar storage. Use of the highly scalable and fault-tolerant distributed streaming, messaging and persistent storage systems make our framework robust and scalable.

B. Query Processing

In order to get data processed by the proposed framework, a Python daemon script need to be executed in addition to the submission of a CQ. The CQ3D *pre-processor* on receiving 3D spatial streams from variety of different sources and in different formats, cleans and integrates them so that uniform data may be provided to the other components of the framework. The *pre-processor* makes use of daemon scripts written in Python to achieve this task. Please note that multiple computing nodes are utilized to speed-up the pre-processing in parallel. The cleaned tuples are then loaded into Kafka topics. *Existing point-cloud operator* gets the pre-processed data from the Kafka topics and perform one or more operation(s) (object detection, segmentation, etc.) followed by loading the resultant tuples into the Kafka topics. As the name indicates, *existing point-cloud operators* makes use of existing point-cloud libraries to perform different operations on cleaned 3D spatial data. Just like *pre-processor*, this component is deployed across multiple nodes to speed-up the processing. CQ registered to the *continuous query processor* gets data from Kafka topics, processes it and generates results which are available to users and applications via API.

Example 2: Assume an MMS vehicle scanning a road segment and generating streams of point-cloud, images and IMU/GPU measurements. The CQ3D *pre-processor* cleans and integrates them and loads them to Kafka topic. After pre-processing, the stream is supplied to the *existing point-cloud operators* which implements an object detection library to detect objects from the pre-processed stream. The Query 1 registered to the *continuous query processor* receives the stream of detected objects and performs a window-based count

of the objects by applying filters on the *object_type* and *prediction_score* attributes. The query implements the distance-based window proposed in Sec. IV of size 2000 meters which slides every 200 meters. The CQ results, corresponding to the size of sliding window, are available to the end-user for analysis through API.

```
windowedCount = predictionStream.groupBy(window
($"detection_timestamp", "2000 meters", "200 meters"),
$"object_type", $"prediction_score").count()
      .filter($"object_type" === "car")
      .filter($"prediction_score" > 0.5)
```

Query 1. A windowed CQ with aggregate and filter operators

VI. Experimental Evaluation

We divide the evaluation section into the following two subsections. 1) Distance-based window and, 2) CQ3D framework.

A. Distance-based Window

1) Data and Experimental Setup: To evaluate the distance-based window, *GeoLife GPS Trajectory dataset* is used collected by Microsoft Research Asia [25]. The GPS trajectory dataset was collected by 178 users during a period of over four years (from April 2007 to October 2011). A GPS trajectory is represented by a sequence of time-stamped points, each of which contains the information of latitude, longitude and altitude. Each trajectory is available as a separate CSV file, which were integrated and supplied as a stream of tuples to the distance window. The distance-window is implemented using c++, which is available as open source on Github[1]. The code is evaluated on a machine running Ubuntu 16.04 LTS OS with Intel Core i7-6700 @ 3.40 GHz CPU and 16 GB memory.

2) Evaluation: First of all, the three distance functions presented in Sec. IV are evaluated. Namely, we compare the famous Haversine distance function with the Equirectangular approximation and the Spherical law of cosines distance functions. Fig. 5(a) compares the relative execution cost of the three distance functions. The execution cost is computed by setting the window size to 3000 meters and passing all the trajectory tuples through the window. Since at any time, trajectories' tuples corresponding to the latest 3000 meters can be kept in the window, for each arriving tuple, corresponding trajectory length (TLength) is computed using the distance functions. Table I lists the number of distance function computations performed by the window to process all the trajectory tuples. From the Fig. 5(a), Haversine seems to be the most expensive, followed by the Equirectangular approximation and the Spherical law of cosines, however from the Table I, the number of distance computations required in case of Equirectangular function for the same amount of data is far higher than the other two. This proves that although the Equirectangular approximation is computationally less expensive than the other two functions however the number of distance computations required is quite high due to its approximate computation. On the other hand, the Havesine and Spherical Law of Cosines can produce accurate results, however the Spherical Law of Cosines is computationally less expensive.

[1]https://github.com/salmanahmedshaikh/DistanceWindow

TABLE I
Number of Distance Computations

	Window Size		
Dist. Function	1000	3000	5000
Haversine	19114852	45383697	65698136
Equirectangular	27550232	62701249	88022883
Law of Cosines	19114852	45383697	65698136

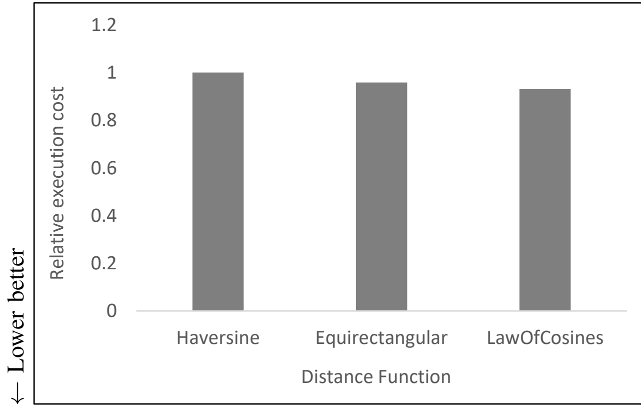
Next we compare the relative throughput of the distance-based window by varying the window size, where throughput can be defined as the maximum number of input tuples processed per second. From Fig. 5(b) it is obvious that the throughput decreases with the increase in window size for all the distance functions. With the increase in window size, the trajectories' length in the window also increases, hence larger number of distance computations are required for the TLength computation, resulting in lower throughput. Another thing to note in the figure is that the throughput is the highest for the Spherical law of Cosines function for all the window sizes, because this function is the least expensive to compute (Fig. 5(a)).

B. CQ3D Framework

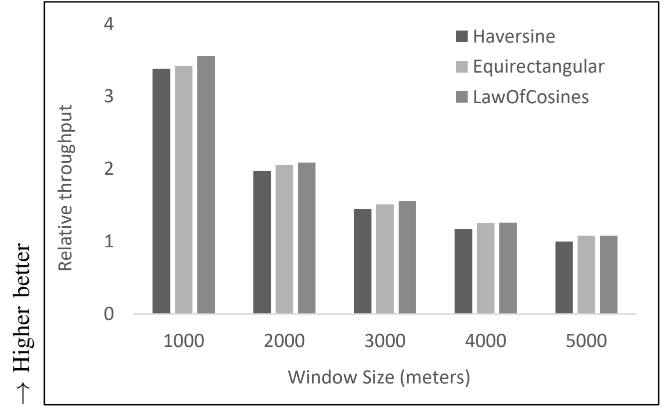
This section compares the proposed CQ3D framework with the traditional framework. The implementation of the two frameworks for the experiments is shown in Fig. 6. The figure divides both the frameworks into three phases, i.e., I. object detection, II. loading detection data and metadata and III. query processing and output generation.

1) Data and Computing Cluster: For the CQ3D evaluation, MMS drive data available from Kitti [1] is used. It is a computer vision benchmark suite containing a number of real 3D spatial datasets collected from MMS vehicle drive. The dataset includes streams of camera images (color and grayscale), laser scans (point-cloud), GPS measurements and IMU accelerations. Since the Kitti benchmark is aimed at object detection, the individual drive data is not so large, hence we integrated all the drive data available from Kitti and supplied that repeatedly to the experimental frameworks for the sake of evaluation.

Since the Kitti benchmark data is available in the form of flat files, it is supplied as files to both the frameworks for the detection of objects (Phase I), without loss of generality, the phase I is equally capable of processing real-time data streams. In the phases II and III, the data is available as stream in the proposed framework while as static data in the traditional framework. Since the traditional approach makes use of distributed database system to store the data, we made use of Apache Cassandra, which is a distributed columnar database and Apache Spark as a query processor. Whereas for the proposed approach, we made use of Apache Kafka as a messaging system and Apache Spark Streaming as a



(a) Distance functions evaluation (Window size: 3000 meters)



(b) Throughput evaluation (by varying window size)

Fig. 5. Distance window evaluation

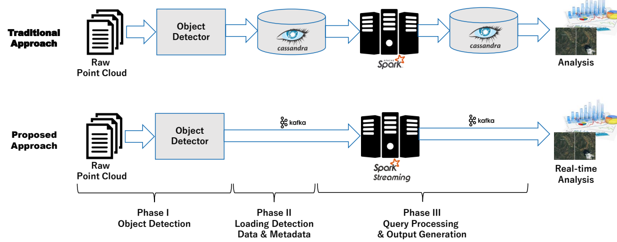


Fig. 6. Experimental frameworks

TABLE II
DEPLOYMENT DETAILS OF EXPERIMENTATION CLUSTERS

Cluster	Nodes #	Memory	Details
Apache Spark	5	256 GB	1 master and 4 worker nodes
Apache Kafka	3	128 GB	1 zookeeper and 2 broker nodes
Apache Cassandra	3	128 GB	1 seed node

query processor. Since the first phase in both the frameworks is implemented in the same way, we only compare the second and the third phases for the execution time. The experiments utilizes 3 different clusters, namely, Apache Spark, Apache Kafka and Apache Cassandra. The clusters are deployed on AIST AAIC cloud [26], where each VM has 20 CPU cores and each core uses Intel skylake 1800 MHz processor. Table II shows the deployment details of each cluster.

2) Queries: For the sake of evaluation three continuous queries, i.e, Query 2, Query 3 and Query 4 are used. Query 2 is a simple filtering query, Query 3 is a group-by query whereas Query 4 is a distance-based windowing query with filtering operator. Since we made use of Apache Spark and Apache Spark Streaming for the continuous queries which have APIs in Scala, Java, Python, etc., the queries are written in Scala programming.

```
filteredOutput = predictionStream
 .filter($"prediction_score">0.5)
 .filter($"object_type"==="people" ||
         $"object_type"==="car")
```

Query 2. Filtering query

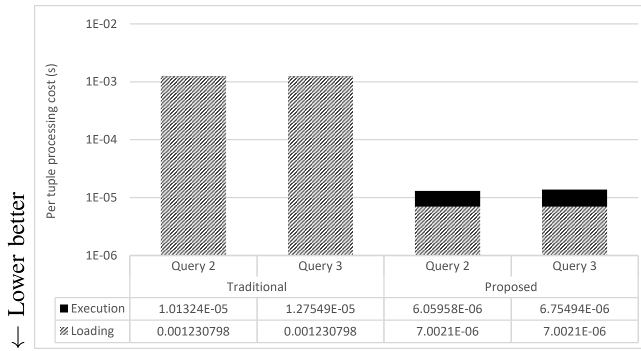
```
groupByOutput = predictionStream
 .groupBy($"drive", $"drive_date",
          $"object_type").count()
 .filter($"object_type"==="people" ||
         $"object_type"==="car")
.filter($"drive"==="0001")
```

Query 3. GroupBy query

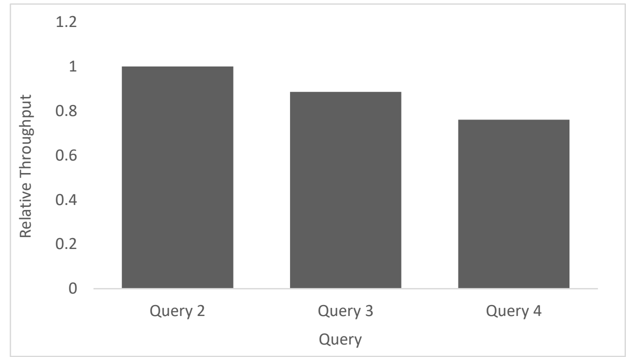
```
windowedOutput = predictionStream
 .groupBy(window($"distance", "100m", "100m"),
   $"drive", $"drive_date", $"object_type").count()
 .filter($"object_type"==="people" ||
         $"object_type"==="car")
 .filter($"drive"==="0001")
```

Query 4. Windowed query

3) Evaluation: Firstly, we compare the proposed CQ3D framework with the traditional framework by utilizing queries Query 2 and Query 3. In this evaluation we compare the data loading time and the query execution time. It is worth mentioning here that the input data is available as either Kafka topic or Cassandra table, where the Kafka topic has replication factor of 2 and is divided into 80 partitions whereas Cassandra table has replication factor of 2. Both the queries takes the same amount of input, i.e., 5,033,350 prediction records/tuples consisting of 13 attributes including "latitude", "longitude", "object type", "detection accuracy" and "bounding box", however producing different number of output tuples, i.e., 116,590 and 44 respectively. The reason for mentioning the number of output tuples is that the measured execution time includes output tuple writing time to the respective storage, i.e., to Apache Cassandra in case of the traditional approach and to Apache Kafka in case of the proposed approach. From the Fig. 7(a), it is evident that the per tuple processing cost of the traditional approach is computationally expensive mainly due to the loading time and the query result writing time. Please note the log scale on y-axis. Per tuple loading time (second phase in Fig. 6) of Apache Cassandra is approx 1.23 milliseconds which is hundreds of time expensive compare



(a) Per tuple processing cost (Traditional vs. Proposed)



(b) Throughput evaluation (all queries)

Fig. 7. Per tuple processing cost and throughput evaluation

to the 7 micro seconds in case of Kafka. This is due to the expensive index building at the time of data loading in case of databases, which is not required in our framework. Besides loading, execution cost per tuple is also slightly higher for traditional approach (third phase in Fig. 6) because the execution time also include query results writing time to the Cassandra DB which is expensive compared to writing data to Kafka.

TABLE III
SAMPLE OUTPUT OF QUERY 4

window (meters)	drive	driveDate	objectType	count
[100, 200]	0001	28	car	2
[100, 200]	0001	28	people	7
[200, 300]	0001	28	car	2
[200, 300]	0001	28	people	8
[300, 400]	0001	28	people	3
[300, 400]	0001	28	car	3
[400, 500]	0001	28	car	3
[500, 600]	0001	28	car	3
[600, 700]	0001	28	car	3
[700, 800]	0001	28	car	3
[800, 900]	0001	28	car	3
[900, 1000]	0001	28	car	1

Next we compare the relative throughput of the three queries, i.e., Query 2, Query 3 and Query 4 in Fig. 7(b). Query 2 has the highest throughput because it is the simplest query with the least number of operators, followed by the group-by query (Query 3) and windowed query (Query 4). The Query 4 is the most expensive query because it contains *window* operator in addition to filtering and group-by operators. The *window* operator causes the group-by operation to be performed for each window. A sample output of Query 4 is shown in Table III. The Query 4 makes use of distance-based tumbling window of size and sliding step of 100 meters each. Table III shows the window-based count of object types ”car” and ”people” by grouping the output with respect to attributes *drive*, *driveDate* and *objectType*. Please note the window size and slide duration, i.e., [100, 200], [200, 300], etc., which are non-overlapping, hence representing tumbling window.

VII. CONCLUSION AND FUTURE WORK

With the availability of low cost 3D scanners recently, 3D spatial data can be obtained easily. The 3D spatial data is generated as continuous data stream, however traditionally it is processed offline, i.e., the data is first stored on a secondary storage, denoised and is then queried and/or analyzed. Traditional approaches to process 3D spatial streams make heavy use of secondary databases to store the intermediate query results, which is IO expensive. Due to this, existing solutions are too slow to support real-time processing of 3D spatial streams. To solve this issue, we proposed CQ3D, which is a main-memory based distributed continuous querying framework capable of processing the 3D spatial data streams in real-time. Furthermore, we proposed a distance-based window to support the continuous queries over spatial data streams, where a user is interested in querying a particular length of spatial trajectories, and implemented it in our proposed framework. For the sake of evaluation, we made use of the real trajectories data available from Microsoft Research Asia and the real MMS drive data from KITTI benchmark suite. Experimental evaluation highlights the drawbacks of traditional frameworks and proves the effectiveness of the proposed framework. Furthermore, evaluation of different distance functions for the distance-based window is also presented. As part of the future work, we are working on an incremental version of the proposed distance-based window to reduce its computation cost.

ACKNOWLEDGMENT

This work was supported by the New Energy and Industrial Technology Development Organization (NEDO), Japan.

REFERENCES

[1] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” *Int. J. Rob. Res.*, vol. 32, no. 11, pp. 1231–1237, Sep. 2013.

[2] A. for Desktop, “What is lidar data?” http://desktop.arcgis.com/en/, [Online; accessed 27-November-2018].

[3] IR, “3d scanning to cg characters in hours,” http://ir-ltd.net/blog/, [Online; accessed 27-March-2019].

[4] P. Greenhalgh, B. Mullins, A. grunnet jepsen, and A. Bhowmik, “Industrial deployment of a full-featured head-mounted augmented-reality system and the incorporation of a 3d-sensing platform,” 05 2016.

[5] P. Oosterom, O. Martinez Rubi, M. Ivanova, and e. a. Horhammer, "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark," *Computers and Graphics*, vol. 49, pp. 92 – 125, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849315000084

[6] PASCO, "Road management that makes use of mobile mapping system (mms)," https://www.pasco.co.jp/eng/products/mms/, [Online; accessed 04-April-2019].

[7] Z. Li, M. E. Hodgson, and W. Li, "A general-purpose framework for parallel processing of large-scale lidar data," *International Journal of Digital Earth*, vol. 11, no. 1, pp. 26–47, 2018.

[8] A. Nath, K. Fox, K. Munagala, and P. K. Agarwal, "Massively parallel algorithms for computing tin dems and contour trees for large terrains," in *SIGSPATIAL/GIS*, 2016.

[9] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using mapreduce," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 31:1–31:42, 2014.

[10] T. A. S. Foundation., "Apache Spark - Lightning-Fast Cluster Computing," http://spark.apache.org/, [Online; accessed 11-November-2018].

[11] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *STREAM: The Stanford Data Stream Management System*. Springer, 2016, pp. 317–336.

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

[13] H. Najdataei and Y. N. et al., "Lisco: A continuous approach in lidar point-cloud clustering," *CoRR*, vol. abs/1711.01853, 2017. [Online]. Available: http://arxiv.org/abs/1711.01853

[14] K. Klasing, D. Wollherr, and M. Buss, "Realtime segmentation of range data using continuous nearest neighbors," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 2431–2436.

[15] J.-S. Kim, K.-S. Kim, and K.-J. Li, "Continuous perspective query processing for 3-d objects on road networks," in *Web and Wireless Geographical Information Systems*, J. M. Ware and G. E. Taylor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–15.

[16] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating window joins over unbounded streams," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, March 2003, pp. 341–352.

[17] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 324–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316719

[18] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 10, pp. 1363–1380, Oct. 2007. [Online]. Available: https://doi.org/10.1109/TKDE.2007.190630

[19] T. A. S. Foundation., "Apache Spark - Lightning-Fast Cluster Computing," http://spark.apache.org/, 2017, [Online; accessed 27-April-2017].

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

[21] Golab, Lukasz, "Sliding window query processing over data streams," 2006. [Online]. Available: http://hdl.handle.net/10012/2930

[22] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches in query processing for moving object trajectories," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 395–406. [Online]. Available: http://dl.acm.org/citation.cfm?id=645926.672019

[23] C. Veness, "Calculate distance, bearing and more between latitude/-longitude points," https://www.movable-type.co.uk/scripts/latlong.html, [Online; accessed 02-July-2019].

[24] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. L. Waslander, "Joint 3d proposal generation and object detection from view aggregation," *CoRR*, vol. abs/1712.02294, 2017. [Online]. Available: http://arxiv.org/abs/1712.02294

[25] Y. Zheng, H. Fu, X. Xie, W.-Y. Ma, and Q. Li, *Geolife GPS trajectory dataset - User Guide*, geolife gps trajectories 1.1 ed., July 2011, geolife GPS trajectories 1.1. [Online]. Available: https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/

[26] N. I. of Advanced Industrial Science and T. (AIST), "Aist artificial intelligence cloud (aaic)," https://www.airc.aist.go.jp, [Online; accessed 4-April-2019].