SpatialSSJP: QoS-Aware Adaptive Approximate Stream-Static Spatial Join Processor

Isam Mashhour Al Jawarneh, *Member, IEEE*, Paolo Bellavista, *Senior Member, IEEE*, Antonio Corradi, *Senior Member, IEEE*, Luca Foschini, *Senior Member, IEEE*, and Rebecca Montanari, *Member, IEEE*

***Abstract*—The widespread adoption of Internet of Things (IoT) motivated the emergence of mixed workloads in smart cities, where fast arriving geo-referenced big data streams are joined with archive tables, aiming at enriching streams with descriptive attributes that enable insightful analytics. Applications are now relying on finding, in real-time, to which geographical regions data streaming tuples belong. This problem requires a computationally intensive stream-static join for joining a dynamic stream with a disk-resident static table. In addition, the time-varying nature of fluctuation in geospatial data arriving online calls for an approximate solution that can trade-off QoS constraints while ensuring that the system survives sudden spikes in data loads. In this paper, we present SpatialSSJP, an adaptive spatial-aware approximate query processing system that specifically focuses on stream-static joins in a way that guarantees achieving an agreed set of Quality-of-Service goals and maintains geo-statistics of stateful online aggregations over stream-static join results. SpatialSSJP employs a state-of-art stratified-like sampling design to select well-balanced representative geospatial data stream samples and serve them to a stream-static geospatial join operator downstream. We implemented a prototype atop Spark Structured Streaming. Our extensive evaluations on big real datasets show that our system can survive and mitigate harsh join workloads and outperform state-of-art baselines by significant magnitudes, without risking rigorous error bounds in terms of the accuracy of the output results. SpatialSSJP achieves a relative accuracy gain against plain Spark joins of approximately 10% in worst cases but reaching up to 50% in best case scenarios.**

***Index Terms*—Algorithms for data and knowledge management, Data Architecture, Spatial databases and GIS, QoS Data Management, Spatial Join, Spatial Indexes, Geospatial Analysis, Apache Spark, Query Processing, Big Data Applications.**

ABBREVIATIONS

AQP	Approximate Query Processing
CV	'coefficient of variation'
DSP	data stream processing systems (DSP)
IoT	Internet of Things
QoS	Quality-of-Service
OOM	Out-of-Memory
PID	Proportional, Integral, and Derivative
SLA	Service Level Agreements
SPE	Stream Processing Engine
SpatialSSJP	Spatial-aware Stream-Static Join Processor
SE	Standard Error

Manuscript received 7 May 2022; revised 17 September 2023; accepted 3 November 2023. Date of publication 6 November 2023; date of current version 27 November 2023. This work was supported in part by the "H2020 SimDOME—Digital Ontology-Based Modelling Environment for Simulation of Materials" EU Project under Grant 814492, and in part by the OntoTrans EU Horizon 2020 Project under Grant 862136. Recommended for acceptance by V. Cardellini. *(Corresponding author: Luca Foschini.)*

The authors are with the Dipartimento di Informatica – Scienza e Ingegneria (DISI), University of Bologna, 40136 Bologna, Italy (e-mail: isam.aljawarneh@studio.unibo.it; paolo.bellavista@unibo.it; antonio.corradi@unibo.it; luca.foschini@unibo.it; rebecca.montanari@unibo.it).

Digital Object Identifier 10.1109/TPDS.2023.3330669

I. INTRODUCTION

NOWADAYS, several businesses massively rely on heavily-trafficked data streaming pipelines for nourishing their backdrops to transform raw data into meaningful information. The abundance of IoT devices almost everywhere, and present in all aspects of our lives, offers giant amounts of fast arriving geo-referenced data streams that serve as sources for an innumerable data-intensive online services. This unprecedented amount of data normally challenges the capacities of state-of-art data stream processing systems, especially when confronted with stringent latency and/or accuracy goals expressed through Service Level Agreements (SLAs).

In addition, most effective dynamic application scenarios, such as those recurring in smart cities, require intermixing various workloads in a mashup fashion [1] by joining data-at-rest with streaming data to pluck required insights. This operation, known as *stream-static join*, where one side of the join is a data stream, while the other is a static file residing in disk or in fast memory, is naturally expensive and may easily turn computationally prohibitive [1], particularly in distributed computing deployments where cross-network data shuffling is normally involved.

In this paper, we focus on scenarios where cluster computing resources are scarce, thus increasing the computation power by overprovisioning resources is not an option, or even with abundance of computing resources, the reason behind the scarcity of resources for a specific stream processing application is that many heavily-trafficked data stream applications could be running at the same time, leading to several processing tasks being executed in parallel. The goal is to optimize the utilization of available distributed computing resources in an adaptive fashion to enable the join processing of massively arriving georeferenced data streams. At the same time, our solution aims at keeping the Stream Processing Engine (SPE) as marginally stable as

possible. This is possible by employing an Approximate Query Processing (AQP) solution. System's stability is guaranteed by relying on a control loop feedback mechanism controller (i.e., PID, will be discussed shortly in Section III.C). It is known in automatically applying a responsive and accurate correction to a control function. It can restore the processing capacity to the desired state with minimal delay by reducing the arriving data size in a controlled manner.

AQP is the most adopted solution for the online information overloading problem [2]. This is so because approximate query responses with rigorous error bounds commonly suffice for strategic decision making in smart city scenarios [2]. For example, generating highly accurate interactive heatmaps from sensors data depending on a geospatially representative well-balanced sample, which is an operation that depends on stream-static geospatial join. Despite being approximate, such a solution is normally adequate for studying the mobility patterns in a dynamic city. AQP systems normally trade off an acceptable tiny loss in accuracy for a lower latency. They do so by operating on a partial subset of the arrival data stream tuples. In a stream-static join setting, accuracy loss occurs because the full static table (e.g., representing regions in a city) is joined with data stream samples, thus from each spatial region being associated with a result computed only over a sample of the data. Latency and/or accuracy goals are normally tuned in AQP systems by an expert user.

However, performing AQP on join results of online geo-referenced data stream samples is typically subject to undesirable degrees of loss in accuracy. This is so because state-of-art SPEs normally embrace randomness by relying on sampling designs that are based on simple random sampling [3]. We consider the AQP on join results because the online aggregation occurs on reduced data stream samples that are fed to the join operator downstream in the pipeline.

The oscillating nature of data streams in the currency of arrival rates and skewness imposes additional challenges that strain the current geospatial join AQP frameworks. Hence, novel approaches should be operating in adaptive modes so that they respond interactively to such fluctuations, aiming always to keep the pipeline operators alive even in the presence of sudden spikes in data arrival rates. In other words, proposals should consider trading off a set of prespecified Quality-of-Service (QoS) goals such as latency, throughput, and accuracy.

To achieve those, this paper presents SpatialSSJP, an adaptive QoS-aware distributed approximate geospatial aware stream-static join processor, specifically tailored for dynamic smart city workloads. By *stream-static geospatial join*, we specifically mean joining geo-referenced arriving data stream tuples with a disk-resident master table that contains enrichment data. SpatialSSJP serves a prespecified set of accuracy/latency goals while processing fast-arriving big geospatial data streams.

More in details, SpatialSSJP encompasses a controller that allows the user to choose between two antithetical goals, namely low-latency and high accuracy. We tightly coupled SpatialSSJP with a spatial-aware sampling method from our previous work called SAOS [3], which selects geospatially-representative well-balanced samples from fast arriving

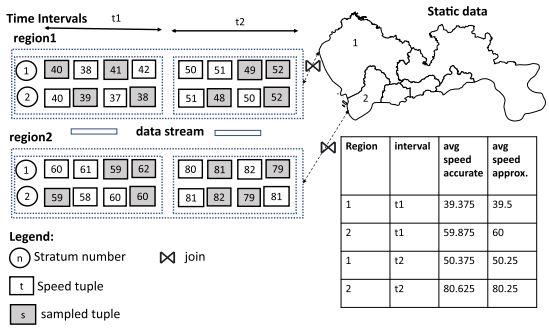

Fig. 1. Hypothetical example: AQP over stream-static join data.

geo-referenced data streams. SAOS is a stratified-like method, which first stratifies (i.e., divides) the arriving stream in groups (each group is known as stratum in stratified-sampling designs), then it places each arriving spatial object in the appropriate group (stratum) based on location similarity with confounding objects in the same group. In other terms, geographically nearby objects end up being placed in the same stratum. By placing SAOS in the frontstage, the system can reduce the number of tuples that will be served to the subsequent stream-static join operator, thus relieving the pressure on the network communication, and improving the processing responsiveness. However, SAOS in its stock version receives sampling fractions from the user interactively. By interlacing SpatialSSJP with SAOS in the frontend, a controller of SpatialSSJP, that is located after the join operator, will interactively and automatically compute appropriate sampling fractions to admit for a subsequent time interval based on robust geo-statistical models. Afterwards, it serves the sampling fractions backward to SAOS for a subsequent sampling round. The selected sample is then served to the geospatial stream-static join processor, which, in turn, performs the join operation. It then sends the intermediate join output to an approximator, which computes an aggregation result over the join intermediate result and serves it, together with a rigorous error bound, to the user in the presentation layer. Since the dataset served to the approximator is a partial subset of the arriving data stream, the result calculated by the approximator is an approximation, and hence is associated with error-bounds. By doing so, SpatialSSJP achieves a set of prespecified QoS goals, either high-accuracy or low-latency, however trading them off efficiently when contradicting. Fig. 1 depicts a hypothetical example answering the following query "what is the approximate average speed of taxis in each region of the city of Shenzhen in China across time".

We prototyped SpatialSSJP on top of Spark Structured Streaming [4] and evaluated it with real-world vehicle mobility geo-referenced Big Data. Queries included geospatial online stateful aggregations that incorporate stream-static join operations within a pipeline. Our results show that SpatialSSJP survives sudden aggressive spikes in data stream arrival rates with various combinations of oscillations in the ranges of '500K to 2000K', '500K to 3000K', '500K to 5000K', '500K to 2000K to 1000K' tuples/second, thus achieving prespecified latency goals. As compared to a baseline without sampling,

which instead either does not survive those fluctuations and easily reaches Out-of-Memory (OOM) exceptions or does not effectively tradeoff latency with accuracy. Also, SpatialSSJP with SAOS in the front-stage achieves various accuracy goal values ranging from stringent (0.01) to more permissive (0.09), outperforming a baseline that employs a Simple Random Sampling (SRS) in the frontstage. By relying on SpatialSSJP, on average, the relative gain over Spark-based native counterpart running online aggregations with join operations and SRS in the frontstage is at least 10% and tipping up to 50%, given the same sampling fractions.

We make the following contributions by introducing SpatialSSJP, 1) an adaptive controller for geospatial stream-static join processing in distributed data stream processing deployments with limited resources (or abundance of resources utilized for several tasks in parallel). The controller serves appropriate sampling rates to a geospatial-aware stratified-like sampler that maintains the quality of geo-statistics over join results, while reducing the data size shuffled over the communication network. We also thereafter transparently incorporated the controller with a de facto SPE (specifically, Spark Structured Streaming [4]), 2) a reactive accuracy-aware procedure, which accepts an accuracy QoS goal and reacts in a way that guarantees achieving the goals by interactively and regularly serving sampling fractions to a stratified geospatial sampler so as to satisfy predefined accuracy goals for approximate geospatial stream-static join calculation, 3) sub-modules that are responsible for incrementally computing approximate online aggregation geo-statistics (e.g., count, average) on stream-static geospatial join outputs. By incremental computation, we mean that the results are gradually building up based on the previous results from the preceding time periods without the need to re-evaluate geo-statistics from scratch. To the best of our knowledge, our framework is unique in the terms of Big Data stream processing applications in the sense that it is the first of its kind that successfully incorporates a stable loop control feedback mechanism to enable controlled AQP on sampled join results of big geospatial data stream. This has a significant implication for a consortium of data-informed urban planning and other relevant smart city applications. This is so because most smart city applications require joining data that is georeferenced and fluctuating in size and arrival rates. We are not aware of any system in the related state-of-the-art that offers similar functionalities.

The remainder of this paper is organized as follows. We first briefly summarize the related theoretical backgrounds. Thereafter, we briefly recapitulate the design and realization of our system. In what follows, we show our findings accompanied by convenient discussions. Thereafter, we discussed some related literature. The paper closes by summarizing the contributions and recommending future research.

II. Problem Statement and Background

Decision makers employ tremendous fast-arriving IoT online data that comes normally from several sources for various kinds of analytics in real-world application scenarios, e.g., average speed of vehicles across regions. Such scenarios require joining IoT geo-referenced data streams with disk-resident data, e.g., to specify to which region each vehicle speed measurement belongs. To meet the user's expectations in terms of system quality, SPEs need to well-balance a prespecified set of QoS goals, most importantly, accuracy versus timeliness. On one hand, timeliness is essential in reflecting a fresh state of the system. For example, computing the average speed across regions during a sudden spike in the number of circulating vehicles needs to be instantaneous to allow for a proper reaction. This could require sampling/dropping part of the arriving stream tuples, negatively affecting the accuracy of the 'average' statistics. On the other hand, higher accuracy goals require more data.

In this paper, we focus on geospatial analytics with geospatial join as a pillar factor that dominates the equation towards achieving accuracy and time constraints.

We particularly aim at addressing questions similar to the following, how an SPE can produce the average speed of vehicles across city regions every minute within predefined error-bounds, over fast arriving geo-referenced mobility data streams that are characterized by being highly-skewed in densities and fluctuating in arrival rates. To address such a challenge, the system should be able to specify the amount of data that need to be discarded while achieving the error-bound and time targets. Based on dynamic prediction of accuracy, load and latency, the data stream processing system specifies the amount of data to sample to meet the objectives of accuracy and latency. A reasonable baseline to compare with our system is a system that either does not sample the data, that would thus be unable to achieve the time-based quality constraint or a system that is based on random sampling, thus unable to achieve the accuracy targets.

Mobility data is normally served in a parametrized format as GPS coordinate streaming tuples (typically longitudes and latitudes). As such, solving a query similar to the aforementioned requires joining those tuples (geospatially) with a master table that contains regions of the city (i.e., neighborhoods in city administrative terms). This kind of join is known as *stream-static join*, where one side of the join is a streaming source, whereas the other is a static disk-resident relation.

Having all that in mind, stream-static join is a potential optimization candidate. This is so because distributed stream-static join runs into several challenges that normally do not affect other computations such as batch-oriented static-static join processing. For instance, data streams normally exhibit peak periods where loads exceed capacities of computing resources.

A. Spark Structured Streaming and Distributed Geospatial Join Processing

As a reference system, we consider Apache Spark [5] and a streaming layer engineered atop of it. Spark is a distributed data processing system that constitutes a cluster composed of one master and several workers.

Spark Structured Streaming [4] (hereafter SpSS for short) is a new layer on top of Spark. Users express queries using a declarative SQL-like API (in the form of DataFrames [6]). The

processing model of SpSS depends on micro-batches, meaning that it processes streams as small batches scheduled to run on Spark parallel jobs. Users write their queries as if they were to be executed in a batch mode and SpSS executes them in a streaming mode. The only difference is that the user must specify the window size over which the accumulated results need to be reported on the presentation layer. This is specifically true for stateful aggregations such as Top-N query types. Stated another way, within each trigger interval, every micro-batch is processed independently as it is considered a miniscule version of subset of batch of data. Aggregation of those micro-batches then resembles the streaming dimension. SpSS semantics are based on incrementalization. In doing so, it treats the stream as an unbounded table, where every arriving tuple is appended to that unbounded input table. User expresses a batch-like query and the underlying SpSS engine translates that into an incremental query scheduled to be executed on the unbounded input table. Results in the result table are updated based on a batch interval. SpSS supports SRS and stratified sampling on micro-batches.

Spark normally performs join operations by either repartitioning or broadcasting. In cases where one of the relations (tables) can reside in the fast-memory of all worker computing nodes (that are forming the computing cluster), the smaller relation (table that is small enough to fit in memory) is broadcasted to all workers, in addition to the join operator. This basically offers local reachability and avoids the cost of shuffling at join query run time. This is attributed to the fact that every partition in each worker node will be locally joined with the smaller table. What remains needed then is combining all local results in the master node to form the final join result. In cases where the master table is big and thereby cannot reside in fast-memory, then a computationally resource-intensive repartition join is required. Reader are referred to [7] for further information describing the mechanism of those two kinds of join operations in Spark.

Conventional join algorithms in Spark core implementation such as sort-merge join are not readily applicable for our case scenarios. Simply put, they are not designed to perform geospatial join for multidimensional data [8]. One of the well-performing algorithms for geospatial join processing is known as filter-and-refine, which is exploited in a recent Spark-based geospatial-oriented libraries such as Magellan.[1] As a utilitarian example, imagining the Earth fattened out (i.e., a two-dimensional representation of the earth on the form of longitude and latitude coordinates, which resembles x/y coordinates in a cartesian coordinate system). An equal-sized uniform grid network is then imposed on this two-dimensional representation so that the view consists of equal-sized grid cells covering the Earth. Each cell has a geocode (e.g., geohash, Google's S2 and Uber's H3). This view constitutes the embedding space (i.e., study area) where geospatial objects (tuples) are withdrawn from. Each georeferenced tuple (on the form of parameterized points in two-dimensional coordinates, typically longitude/latitude) in the database is hashed with the same geocoding scheme, reducing its multidimensional representation into a corresponding single-dimensional geocode. However, this representation is approximate as it generates some false positives where the coding is similar to that of one of the cells in the grid network, while the tuple falls outside in real geometries. Having this in mind, the filter-and-refine approach proceeds as follows. It first filters data based on a quick-and-dirty sieve, where a cheap equi-join is performed on the single-dimensional geocoded data with the grid cells, which results in false positives. Afterwards, a refinement step is applied, where a computationally expensive geometrical operations are performed (e.g., intersects, within) to test which of the remaining candidates fall in the grid cells in real geometries. It is worth mentioning that those geocodes are the join keys. Readers are referred to [1] for detailed information exhibiting how this method works. Given the forementioned overarching traits of the filter-and-refine approach, we are adapting it in our system design in this paper.

[1] [Online]. Available: https://github.com/harsha2010/magellan

B. Geospatial Approximate Query Processing

Approximate Query Processing (AQP) is a computing field that is concerned with cases where exact solutions are either not necessarily needed or too expensive to put into action [2]. It depends on serving approximate results with rigorous error-bounds expressed, for example, in terms of confidence intervals. What makes AQP significant is the observation that users normally accept to forego tiny accuracy loss for the sake of a high speedup gain [2], [9]. In addition, policy makers typically can make discernibly accurate decisions even while query responses are not perfectly accurate. For example, interactive heatmaps are normally used for decision making without being perfectly accurate.

AQP relies on reducing the input data size using several approaches such as spatial sampling [3], [10]. To achieve accuracy targets, sample should be well-representing the population from where it is withdrawn, taking into considerations the nature of the data being processed. Spatial information from close-by measurements is generally not accounted for in classical sampling theory, thus other sampling designs that are based on stratification are normally favored as they prove to yield better estimator values for population target variables.

Our stratified-like sampling method called SAOS (in addition to its extended version, ex-SAOS) that we have designed in our previous works [3], [10] is an important contribution for spatial AQP. SAOS algorithm is analogous to the following heuristic overview. Imagining the earth in its two-dimensional planar geometrical space (analogous to a rectangular grid), SAOS first overlays the study region with a square-shaped grid. Thereafter, it selects randomly a spatially-proportional unbiased number of spatial points from each grid cell independently. Arriving geospatial data stream tuples are multidimensional parametrized pairs on the form of longitude/latitudes locational pairs. The system first encodes each tuple with a geocode (specifically geohash, resembling a Minimum Bounding Rectangle (MBR) that represents the approximate location of the arriving tuple). Those geocodes correspond to matching geocodes that represent the embedding space (the space from which tuples are withdrawn). By simply selecting tuples from each geocode bracket independently, SAOS ensures that the same percentage is selected from each region in the embedding space. This is so because each region is represented by the covering geocodes. SAOS is a

stratified-like sampling method, which makes it preferable over other geospatial methods that are based on random sampling. This is so because SAOS selects representative input tuples fairly from each region in real geometries so that no region is overlooked. It worth mentioning that SAOS is built with Spark's 'filter' and 'map' transformations, where a geocode is generated for every arriving parametrized georeferenced data stream tuple using a simple 'map' transformation, thereafter a 'filter' transformation is applied to select tuples randomly based on the sampling rate. Interested readers are referred to our previous works [3], [10] for further information on SAOS. Since we rely on a filter-and-refine approach for spatial join in this paper, then SAOS is a perfect match in combination with these kinds of geospatial join methods. This is so because SAOS, by design, selects a fair number of tuples from each grid cell in the study area. Since each grid cell has a unique geocode, which is the join key, then SAOS guarantees that no single join key will be overlooked. This, in turn, guarantees preserving the geo-statistical properties of the approximation over the join output.

III. SpatialSSJP Overview

SpatialSSJP (short for Spatial-aware Stream-Static Join Processor) is a novel adaptive QoS-aware system for performing distributed geospatial stream-static joins efficiently on massive amounts of geo-referenced data streams. SpatialSSJP accepts a georeferenced input data stream consisting of data items arriving from various mobility IoT sources, which needs to be joined with disk-resident static enrichment data. It also receives an expert-guided continuous geospatial aggregation query, which implicitly includes a geospatial stream-static join operator. In addition, it receives a query running budget expressed as guarantees on either a latency or an accuracy QoS goal. SpatialSSJP then guarantees that the data stream is processed within the query running budget. It does so by relying on AQP and operating only on a partial set of the input tuples from the arriving geospatial data stream and publishing regularly an approximate incremental output with rigorous error bounds.

Basically, we have hybridized a novel data rate controller with our geospatial-aware sampling method SAOS [3]. SAOS receives the proper sampling rate that is communicated by the rate controller of SpatialSSJP. It then selects a partial subset of the arriving data tuples so as to satisfy the query running budget, then it serves them to the subsequent geospatial stream-static join operator. Afterwards, SpatialSSJP applies the geospatial approximate aggregations operators on the join output and serves incremental results to the user. SAOS assures that the sample selected is fair and realistically reflects the geospatial distribution of data in real geometries. The context diagram of Fig. 2 schematically shows a high-level overview of SpatialSSJP, which encompasses three main components: *stream-static join processor*, *rate controller* and *geospatial approximator*.

A. System Model

State-of-art distributed data stream processing systems (DSPS) are categorized into two types: (i) micro-batch-based models, and (ii) record-at-a-time stream processing. Even though the latter is known to result in lower latencies for several deployment settings, we avoid employing it in this paper. This is so because our sampling method is a stratified-like sampling method, which needs first to stratify the arriving tuples. If employed to record-at-a-time model, then the so-called reservoir sampling would be required, to retain the first part of the data that arrives, then stratify it and apply the stratify-like sampling. This would introduce extra complexities which would counteract the benefits of sampling and approximate computing consequently.

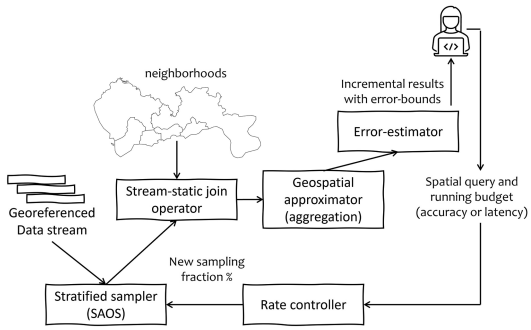

Fig. 2. SpatialSSJP architectural overview.

Micro-batch data stream processing model processes data streams in series of small batch tasks, achieving second-scale end-to-end latencies that are sufficient for smart city scenarios [11]. An SPS that operates with micro-batching checks regularly streaming sources, then it executes batch queries on new data arrived after last batch. Continuous processing (record-at-a-time) models fetch records continuously from streaming data sources, thus reducing latency to milliseconds and satisfying low-level latency requirements.

In this paper, we choose to operate on a micro-batching mode instead of a record-at-a-time model for a very important reason. Our sampling method is based on stratification. As such, it first needs to stratify the micro-batch before selecting. Performing sampling in a record-at-a-time model runs into several complexities that are not confronting micro-batch modes.

B. System Design Assumptions

We have designed SpatialSSJP with the following set of design assumptions.

The query running budget is set by a domain expert. The budget is expressed in terms of either required accuracy or latency. Accuracy running budget is expressed as a value that equals to the required "margin-of-error" as will be explained shortly in Section III.D, and latency running budget is expressed as time in milliseconds, representing the target scheduling delay and processing time. Our system then guarantees that the received data streams are processed within the specified query budget boundaries, either accuracy or latency after few batch intervals depending on the aggressiveness of the spike in the data arrival rate. Even though achieving the contradicting accuracy and latency targets would be impossible by any data stream

processing system, our system strikes a plausible balance between both whenever achieving both altogether is not possible. It should be noted that in data streaming environments, depending on the aggressiveness of the spike in the data arrival rate, it could be very challenging to achieve latency/accuracy targets during the first batch interval that witnessed the spike. Our system instead works stepwise where accuracy improves on a batch-after-batch basis during the spike (which normally extends for many batch intervals) until it eventually achieves the required accuracy/latency targets. In this sense, our system "eventually" guarantees achieving the prespecified latency/accuracy targets.

In this paper, we consider window-based aggregations. We specifically focus on tumbling windows, which is a series of contiguous non-overlapping fixed-sized time intervals, where an arriving tuple can only be assigned to a single window. Another type of windows includes sliding windows, where windows overlap, and a tuple can be assigned to multiple windows. We scope ourselves to tumbling windows in this paper and consider extending the system so that it supports sliding windows as a future research work, as sliding windows can run into more complexities. More in detail, even though sliding windows are 'fixed-sized', in cases where 'slide duration' is smaller than 'window duration' input tuples are bound to multiple windows. This means that stateful aggregation computations are repeated several times, adding its toll to the running times.

Also, we consider a deployment setting where a mid-range business can deploy a Cloud public cluster or in-premises cluster with only a few computing machines (e.g., on the par of 4 to 6). Thus, provisioning extra resources is not an option, mainly due to the cost and budget constraints of the business operations.

It worth mentioning, though, that the current version of the system supports only stream-static join. Thus, stream-stream join is not supported. This is so because stream-stream join runs into challenges that are not normally present in a stream-static counterpart [12]. To clarify, at any point of time, the view of both sides of the stream-stream join (the two data streams) is incomplete, which makes it challenging to find matches between the two data streams. Any tuple from one of the streams can potentially match with any upcoming (have-not-yet-arrived) tuple from the other stream. This has an implication on making it harder to decide which tuples to drop from each individual data stream, even with a knowledge of sampling rates. Another reason is that we consider cases where the static table cannot be loaded entirely to main-memory. For example, tables storing geographical representations on the form of polygons for a metropolitan city such as NY City in USA are relatively large. Having said that, we leave the tailoring of our system so that it supports stream-stream scenarios as a future research perspective.

C. System Workflow

The main operational design goal of SpatialSSJP is to minimize the number of geospatial data stream tuples that need to be served to a stream-static join operator in DSP systems. In other terms, the system works on dropping a partial subset of the arriving geospatial data stream tuples in a way that guarantees obtaining gains in the processing speed without negatively affecting the geo-statistical accuracy.

To achieve those design goals while responding to a user query requesting online aggregation over geospatial join outputs, SpatialSSJP operates as follows. In the upstream, geo-referenced data is continuously ingested and served at regular time intervals. The first component that data confronts is SAOS in the front-stage. SAOS is an always-on component waiting for a signal from the *'rate controller'* of SpatialSSJP, which is responsible for deciding upon the correct sampling fraction to be withdrawn during subsequent batch intervals. It specifically employs procedures to calculate the correct sampling fraction that satisfies the user's query running budget. SAOS in this way acts as a filter that minimizes the size of the input data stream that needs to be transferred toward the join processor. Depending on the sampling fraction, SAOS selects a sample from subsequent batch interval and serves it to the pipeline downstream that is encompassing a *'stream-static join processor'*. The join processor then runs a parallel job to execute the stream-static join part of the user's query on the sample, and then sends an intermediate result to a *'geospatial approximator'*. Since SAOS is geospatial-aware, and because it selects a fair number of tuples for each join key independently, then the intermediate join output maintains the desired geo-statistical properties. The geospatial approximator then updates the online aggregations (expressed as part of the user's online spatial query) and serves up-to-time fresh results to the user in the presentation layer. The system also employs a computational model to calculate the error-bounds and serve them with the output. Simultaneously, the *'join processor'* reports the latest geo-statistics to the rate controller, which uses them for adaptively computing a new sampling fraction and feeding it back to SAOS in the front-stage, and so on and so forth.

D. Design Details

The main operational design goal of SpatialSSJP is to minimize the number of geospatial data stream tuples that need to be served to a stream-static join operator in DSP systems. In other terms, the system works on dropping a partial subset of the arriving geospatial data stream tuples in a way that guarantees obtaining gains in the processing speed without negatively affecting the geo-statistical accuracy.

The main component of our system is the *rate controller*. The workflow of the 'rate controller' component is listed in Algorithm 1. An interface at the presentation layer provides the ability for user to input QoS targets as either a desired latency or a desired accuracy. Our rate controller is designed so that it then guarantees that an end-to-end stream-static join (being part of a DAG) completes running within the prespecified QoS budget. By doing so, it computes a convenient sampling fraction relying on one of two procedures depending on the user's join QoS goals (latency or accuracy). Despite relying on different sets of theories, depending on the user's join query budget, both procedures compute and serve a sampling fraction back to SAOS based on the user's geospatial join requirements, which

Algorithm 1: rateController Procedure

Input: latency, throughput, OR accuracy (a.k.a. margin of error) targets
1: **Procedure** rateController (latThrAccTargets)
2: **If** (priority = = latency)
3: $rate_{new}$ = LatencyAwareControler (latencyTarget, PIDvalues)
4: **Elseif** (priority = = accuracy)
5: $rate_{new}$ = AccuracyAwareController(marginOfError)
6: **End if**
7: **Return** $rate_{new}$
8: **End procedure**
9: **Procedure** LatencyAwareController(latencyTarget, PIDvalues)
/* *retrieving statistical information from the last batch interval, specifically, scheduling delay, Processing time, and number of elements* */
10: lastTriggerInformation = retrieveLastTriggerInfo()
/* *adapted, retrofitted and repurposed from Spark Streaming* [11], [13], *but here applied to sampling (instead of the plain application to backpressure) using SpSS*/
11: $rate_{new} = rate_{latest} - (p.err) - (I.\ err_{hist}) - (D.err_d)$
12: **End** Procedure
13: **Procedure** AccuracyAwareController (marginOfError e)
14: $rate_{new} = z^2_{\alpha/2} v / \boldsymbol{e^2_{des}}$
15: **End** Procedure

guarantees completing the join query within the query budget while serving incremental fresh results.

1) Latency-aware Rate Controller: SpatialSSJP accepts a latency QoS target as a query budget to run an approximate online aggregation over a geospatial join query. It then employs a retrofitted and repurposed version of a well-performing stable controller from the *loop-feedback mechanism theory* known as the PID controller. It computes an error value by simply subtracting a variable known as a 'measured process' variable (or PV for short) from another variable that is known as a desired setpoint (or SP for short). PID controller then enforces a correction depending on three terms known as *proportional*, *integral*, and *derivative*. The process settles the PV variable by minimizing three error values. In our deployments (similar to the way it has been applied to Spark Streaming backpressure version [11], [13]), the 'proportional' term defines how correction depends on the present error (w.r.t. the latest measurement from the latest batch interval information). 'Integral' term specifies the way that the correction should react to the accumulation of historical errors (i.e., accumulated through past batch intervals). The purpose of this term is to speed up the healing process (i.e., the movement towards the desired setpoint SP). The derivative term specifies how the correction depends on the prediction of future errors based on error change between two batch intervals (i.e., the trend).

To our knowledge, by the time we wrote this, backpressure by employing the PID feedback controller has never been implemented in Spark Structured Streaming [4]. It also has never been incorporated with a geospatial sampler for appropriately shedding loads while achieving incremental approximate online aggregation results over geospatial join outputs with high accuracy and low latency as prespecified in the user's query running budget. To resolve this issue, we have retrofitted and repurposed the stock version of the PID rate controller (the version that is termed PID rate controller in Spark Streaming and which has been employed in Spark Streaming [11] for triggering backpressure [13] effect). Our retrofitted and repurposed version is transparently incorporated atop the SpSS layered stack for triggering a geospatial AQP aggregations over stream-static join outputs, within the layers of a declarative API (SQL-like DataFrames API in Spark Structured Streaming [4]). For computing the three constituent terms that are comprising the PID (Proportional, Integrative and Derivative terms), we have exploited an analogous mathematical model-based method similar to the one that has been employed in the plain Spark Streaming version. That said, after each batch interval, the new desirable rate is computed using (1).

$$rate_{new} = rate_{latest} - ((P.err) + (I.\ err_{hist}) + (D.err_d)) \tag{1}$$

where $(P.err)$, $(I.\ err_{hist})$ and $(D.err_d)$ are the proportional, integrative, and derivative terms in the PID controller, respectively. The approach through which every term is computed is adapted from the plain PID application as it appears in [11]. The novelty of our work in this paper then is the ability to successfully incorporate the PID controller transparently within the layers of SpSS for triggering geospatial AQP aggregations over stream-static join output. Since our latency controller resides one step after the stream-static join operator, it can efficiently capture the lateness caused by the fluctuations of the number of tuples that arrive at the join operator. Thereafter, it sends a signal to the sampler, informing it about the appropriate sampling fraction for the next time interval to keep the join operator working in a stable state, without staggered tardiness.

2) Accuracy-Aware Rate Controller: If the user prioritizes accuracy as a QoS goal in the 'online aggregation query over geospatial joins', then our rate controller triggers the accuracy-aware sub-component. This component aims at calculating a new sampling fraction that will achieve the target error-bound (normally expressed as a 'margin of error' QoS target in the user geospatial join query budget). We rely on our SAOS spatial sampling method [3] in the frontstage for selecting a partial subset from the arriving streaming tuples, which resorts to a stratified sampling design. As such, we depend on the theory of stratification [14] for calculating an estimate for an acceptable sampling fraction based on a 'margin of error'. We specifically apply (2).

$$n = z^2_{\alpha/2} v / e^2_{des} \tag{2}$$

where e_{des} is the desired error expressed as a QoS goal and $z_{\alpha/2}$ is the upper $\alpha/2$ point of normal distribution. We calculate v by using (3).

$$v = \sum_{h=1}^{H} {}^{n}/_{n_h} (N_h/N)^2 \, S_h^2 \tag{3}$$

where n_h is the sample size at each moment in stratum h. n is the total sample size (in all strata), N_h is the stratum population size, S_h^2 is the variance of stratum h, N is the population size. All those values are calculated incrementally by our support.

This approach presumes that we have a proper value for v. For example, as a result of a preceding sampling survey. As this may not potentially be the case in streaming settings, we otherwise depend on incremental calculation and loop feedback mechanism in improving the value of v after each batch interval and feeding it back to the accuracy controller. For a 95% confidence level, $z_{\alpha/2}$= 1.96 applies. Thus, we apply (4).

$$n = 3.84 * \left(v/e_{des}^2\right) \tag{4}$$

to calculate the new sample size, where e_{des} is the desired 'margin of error' expressed as a QoS goal within the user's geospatial join query requirements. Since the accuracy-aware controller is placed after the stream-static join operator, it can factor the degradation in geo-statistical accuracy that causes a lag behind the prespecified user accuracy target, and which is caused by a probable low size of the input sample that is served to the geospatial join operator. Based on that, it serves the new sampling fraction to SAOS to achieve the accuracy target prespecified in the join query. Since it serves only the required fraction, it can achieve a plausible balance with the latency target.

It is then apparent that by combining the two equations, we obtain a tradeoff equation between latency (incorporated within the three terms of PID) and 'margin of error' (i.e., accuracy or 'estimation quality'). That is to say (5) applies,

$$rate_{new} = rate_{latest} - ((p.err) + (I.err_{hist}) + (D.err_d)) = 3.84 * \left(v/e^2\right) \tag{5}$$

rendering a closed-form solution that achieves all QoS goals intractable.

3) Geospatial Stream-Static Join Operator: Geospatial stream-static join processing is expensive in distributed settings. This is due to the associated cost of network communication during data shuffling process. Hence, in Cloud deployments with limited resources, data needs to be reduced to cut network cost.

SAOS selects a fair number of data stream tuples from each geocode key independently without overlooking any join key. The other side of the join is a disk-resident master table that contains the regions of a geographical study area. Each region is represented by few keys geocoded with the same hashing mechanism that is applied for the data stream tuples (geohash geocoding in our case). As a running example, suppose we have the following arriving tuples (g1, v1), (g2, v1), (g2, v1), (g1, v2), (g1, v3) and (g1, v4). The corresponding join pairs in the disk-resident side are (r1, g1), (r1, g2) covering one region of a geographical area, where g is the geocode, v is the value associated to the data stream tuple, and r is the region. If the sampling fraction is 0.5., SAOS will select 50% from each join key from the arriving data tuples, say (g1, v1), (g2, v1), (g1, v2). This means that at most 50% of the arriving tuples will be shuffled, and the costly inner join will be performed 3 times instead of 6, resulting in (g1, (r1, v1, v2)) and (g2, (r1, v1)) as the join output. SAOS specifically operates a step behind the filter stage of the filter-and-refine geospatial join that we are retrofitting. Then the refinement is performed over the inner join output. This operational mechanism constitutes the retrofitted approximate version of the plain filter-and-refine approach, which we call geospatial stream-static join operator. The join output will then be served to the geospatial approximator component in SpatialSSJP, which is then responsible for computing an approximate incremental online geospatial aggregation result and serve it interactively to the presentation layer.

In this paper, our geospatial approximator is a retrofitted version of an approximator that we have designed in a previous work [3], where we depend on the theory of stratification [14] for calculating aggregation geo-statistics such as 'average' and 'count'. The geospatial approximator depends also on the theory of stratification [14] for calculating the error bounds. The difference in the current work is that we calculate those geo-stats over the geospatial join output as opposed to the plain method from our previous work [3], which otherwise calculates the geo-stats on the data stream tuples directly. In a more precise sense, the current geospatial approximator is more appropriate for stateful online aggregations over geospatial stream-static join results, thus serving more advanced analytics in urban planning and smart environments. An example stateful spatial online aggregation query is the following: "find the average speed in each region in a city and sort them in descending order". The state that needs to be maintained throughout this query is the average speed in each region key, which changes stepwise as new streaming data pours into the system. Since the arriving data streams are big, implementing the solution in a distributed system guarantees achieving time-based QoS requirements as each worker node computes the average speed for a few regions in parallel.

Since the key for the data stream tuples is a single-dimension geocode, those tuples will be distributed by a hash-driven partitioner, so geometrically nearby tuples have more chances to end up in the same partitions in the underlying distributed DSP system worker nodes. It worth mentioning that geographically nearby objects share geocodes (geohash in our case) that have similar prefixes, hence a hash-based partitioner, typically, will forward nearby objects to the same partitions of the distributed DSP system. Consequently, less data shuffling is involved, which improves the time-based goals of the distributed join. More in details, each worker node of the deployed computing cluster will perform part of the stream-static join on a subset of the selected data sample. Partial results will then be merged by our approximator in the master node and the result will be served interactively after each time interval.

IV. Prototype and Implementation

A. Implementation Insights and Baselines

We prototyped SpatialSSJP by introducing few tweaks atop Spark Structured Streaming (SpSS) [4]. SpSS creates DataFrames from input data stream sources and operates on them incrementally as new data arrives. DataFrames [6] is the abstraction for representing structured data in SpSS, which

offers a user-friendly interface for programmatically expressing relational queries. We offer an interactive simplified interface, based on the DataFrames API, to facilitate expressing user's approximate geospatial online aggregation queries. Technical functionalities of SpatialSSJP are transparently incorporated within the bottom layers of Spark's codebase. This includes the sampling, join, rate controller and approximator modules. To realize this design model, we implemented new functions atop SpSS to perform the 'approximate online aggregation' over a 'geospatial stream-static join' query within the prespecified query running budget over sampling inputs that are served by the sampling module on the form of Spark's micro-batch DataFrames [6].

Our implementation consists of three main modules, in addition to SAOS module, which we adapt from our previous work [3]. The implementation details of SAOS are described in our previous work [3].

The rate controller module consists of two submodules: latency and accuracy controllers. They are both responsible for translating the query budget into an appropriate sampling fraction that will be served to the sampling module. We added two functions over SpSS to compute the sampling fractions using (1) and (4) that are described in Section III.B. Those functions include a procedure for triggering the loop feedback mechanism to regularly re-compute the sampling fractions as a response to the fluctuations in the data arrival rates as described in Section III.B.

For the join processor module, we retrofit and repurpose a geospatial-aware library atop Spark known as Magellan.[2] Specifically, the plain implementation depends on the stock version of the filter-and-refine approach. We implemented a function onto SpSS to couple SAOS with the filter stage so that we maintain its geospatial characteristics. Afterwards, the refinement stage functionality proceeds the operation over the join filter-stage output, as per the plain implementation.

Additionally, as a baseline to compare with our latency controller, we have retrofitted a version of geospatial join in Spark's Magellan so that it works on geospatial stream-static join without sampling. Having in mind that basic functions that perform static-static geospatial join are offered over-the-counter by the base distribution of Spark's Magellan. However, further functions demand extra libraries. We have thus engineered a layer over Spark's Magellan so that it performs the desired functionality transparently.

Our geospatial approximator depends heavily on the incremental computation that is offered by the underlying Spark's codebase. However, the stock version does not offer functions for appropriately approximating geo-statistics. To close this void, our geospatial approximator module adds novel functions that compute geo-statistics incrementally.

As a baseline to compare our system against for the case of the ability in achieving accuracy QoS goal, we transparently incorporated a model-based controller on top of Spark Structured Streaming [4]. The baseline is based on the Simple Random Sampling (SRS) theory [14] as a pre-join sampling method. It computes a new sampling fraction after each batch interval and feeds it interactively to front-stage SRS-based sampling module (as opposed to our SAOS sampling module [3]), which precedes the join operator. SRS does not guarantee fairly selecting distinct join keys, as some keys are overlooked. In other terms, some join keys could be underrepresented in the selected sample, thus resulting in deteriorated geo-statistical properties.

More in detail, for a fair comparison, as we are comparing the employment of SAOS in the front-stage as a pre-join sampler against a random sampling design counterpart. We also depend on the theory of simple random sampling (SRS) [14] for estimating an appropriate sample size based on a target 'margin of error' in cases that SRS is applied instead of SAOS as a pre-join sampling method. We specifically employ (6),

$$n = n_0/(1 + (n_0/N) = 1/(1/n_0 + 1/N) \quad (6)$$

to calculate the desired sample size, where n_0 is calculated using (7).

$$n_0 = z^2\sigma^2/e_{des}^2 \quad (7)$$

We implemented a function onto SpSS to compute the sample size based on (6), which will be served to the SRS-based sampling module in the frontstage. The other modules (join, approximator) then proceed with their operation as described before.

B. Supported Queries

We support online geospatial aggregation (first proposed by [15]), where join is part of the query plan. Since we are operating on window semantics, aggregations typically include some statistic such as an 'average' estimator of an attribute value during each time window [16].

An expert specifies a tolerable error. Those are normally expert investigators in a geo-statistic study who can specify the precision needed, expressed often as in (8).

$$\mathrm{P}\left(|\bar{\mathrm{y}}_{samp} - \bar{\mathrm{y}}_{\mathrm{pop}}| \leq \boldsymbol{e_{des}}\right) = 1 - \alpha \quad (8)$$

where $\bar{\mathrm{y}}_{samp}$ is the estimate of the 'average' value using the sample, $\bar{\mathrm{y}}_{\mathrm{pop}}$ is the estimate of the 'average' using the population, and $\boldsymbol{e_{des}}$ is the permitted error (i.e., margin of error). The investigator normally decides acceptable value for α and $\boldsymbol{e_{des}}$. For example, $\boldsymbol{e_{des}} = 0.02$ and $\alpha = 0.05$ (equivalent to a confidence level 95%) are common. This is equivalent to defining a maximum permitted difference between an estimate (e.g., 'average' or 'mean' of a target variable) and a true value, together with an allowable tiny probability α for the error to exceed the difference, the goal is then choosing a sample size that achieves the equation.

C. Quantifying Uncertainty

To quantify the uncertainty associated with applying the approximate computing instead of the deterministic closed-form solution, we depend on a set of equations adapted from the theory of statistics [14].

[2][Online]. Available: https://github.com/harsha2010/magellan

For single queries, we rely on 'coefficient of variation' (CV) [14] as a measure of relative variability using (9).

$$\widehat{CV} = \frac{SE\left(\bar{Y}_{SAOS}\right)}{\bar{Y}_{SAOS}} \tag{9}$$

which is then equivalent to the standard error (SE) as a percentage of the 'mean', where $SE(\bar{Y}_{SAOS})$ is the standard error of the 'mean' estimation on a sample selected by applying SAOS sampling method. In addition to those, we calculate the gain of applying SAOS [3] (instead of the baseline, which is the SRS sampling design) as a pre-join sampling method residing before the geospatial stream static join operator. We specifically use the 'design effect' (abbreviated deff) [14] as in (10),

$$\begin{aligned} \text{deff} &= gain_{SAOS} \\ &= \frac{estimated\ variance\ from\ SAOS}{estimated\ variance\ from\ SpSS-based\ SRS} \end{aligned} \tag{10}$$

which provides a measure of the precision gained or lost by using a more complicated sampling design instead of an SRS. Our intention is to factor the effect of choosing the appropriate sampling design (as a pre-join reduction method) on the accuracy of the aggregation over the stream-static join. It is true that all sampling methods reduce the number of data stream tuples that need to be transferred and shuffled for the geospatial join to complete. However, they significantly differ in terms of the accuracy of the approximate over the join output as will be shown in the results discussion, where our stratified-like design has a utility in balancing the tradeoff between accuracy and latency.

V. Performance Evaluation and Results

A. Deployment Settings, Test Cases and Benchmarking

Dataset: For benchmarking, we use two mobility datasets. The first dataset is the NY City taxicab trips datasets [17], from which we choose a cohort of six months dataset (around nine million units) representing data captured through taxi rides for the first half of 2016. We choose the green taxi trip records, where a single data point is a JSON payload which include interesting fields capturing, most importantly, pick-up/drop-off locations and trip distances. The static data is represented by a GeoJSON file containing the set of polygons covering NY City in the USA. The second dataset, we use a vehicle mobility dataset that consists of around 1155K tuples, representing Electric Taxi GPS mobility trips for one day in the Chinese city of Shenzhen [18]. The static table in this dataset is a GeoJSON file containing the set of polygons covering the Shenzhen city in China.

Usage Model: while the main purpose of parallelizing the operation of SPEs is to achieve low latency and high throughput, there are innumerable scenarios in highly dynamic and scalable applications that require joining fast arriving geo-referenced data points with static information (i.e., information that is held in disks). This requirement can be difficult to achieve and can have a negative impact on the overall performance of the SPE simply because of the cost incurred by the I/O [19].

An example scenario is the following. NY taxicab trips have been made publicly available in a format that preserves privacy. That said, only GPS parametrized longitude/latitude coordinates are revealed without the titles of the regions to which those itineraries belong. To complement this, names of regions (i.e., neighborhoods or districts in city administrative terms) are supplemented separately in a disk-resident master table. Zones are represented as polygons, where each polygon is represented by multiple points forming the vertices. An online continuous query may then request to "generate an interactive heatmap that shows trajectories of NY taxis as they flow around in the city" aiming to inform some decisions that are related to city planning. An inherent problem in this case is that since points are served as multidimensional data on the longitude/latitude formats, specifying the polygon to which every point belongs demands applying a costly geospatial stream-static join operation. In a streaming environment, those operations require joining geo-referenced data stream tuples with stationary master data. At times, the size of data streams can be exhaustively massive. This stands as an obstacle on the way to absorb the whole population by one computer screen (e.g., for generating heatmaps). It is then sensible to pick slices of the input data and join them with the static master table, relying then completely on the geospatial AQP theories. Due to unpredictable nature of the data stream arrival rates and skewness, it is essential for DSPs to employ adaptive models that respond interactively and proactively, aiming at catching up with the data stream oscillation nature.

Deployment and Experimental Settings: We deploy SpatialSSJP on a Microsoft Azure HDInsight cloud Cluster hosting Apache Spark (version 2.2.1). Our cluster consisted of 6 NODES in total (2 Head, analogous to master nodes in Amazon, plus 4 worker nodes). Head specifications are based on (2 x D12 v2), and workers are based on (4 x D13 v2) specifications. Every head node hosts 4 CPU cores with 28 GB RAM on each and 200 GB Local SSD memory, and quantities are double those figures for each worker node.

Metrics: For benchmarking, we evaluate SpatialSSJP using two metrics: end-to-end latency and accuracy loss. Specifically, the latency is defined as the time between the arrival of data to the stream processing system within a time window and the time a stepwise aggregation result is produced and served to the user in the presentation layer.; To corroborate results associated with the ability of the system to achieve the latency targets while trading off plausibly the accuracy targets, we apply the 'coefficient of variation' (CV) as it appears in (9) of Section IV.C. We also measure the 'average number of keys updated' by using different methods and align the comparison discussion with the CV values for each method.

In other terms, the percentage of the deviation of the estimated 'mean' from the exact mean value. At various experiments, we also measure the sampling rates that are required to achieve the latency or accuracy constraints at each time step (known hereafter as batch ID).

Testing Scenario: We have designed few recurrent mix workload scenarios that necessitate geospatial stream-static join processor to be applied as an integral part of the workload. We

aim at measuring the following, 1) *The ability of SpatialSSJP to meet target latency goals* through applying our latency-aware rate controller. We have applied the same settings to the two methods, SAOS [3] and SRS-based sampling baseline, which are operating in front stages ahead of the join operator. Thereafter, we compare them both. We apply two widely adopted PID value settings for this scenario. In the first setting, we use values P = 1, I = 1, D = 1. For the second setting, we use values P = 1, I = 0.6, D = 0.2. By doing so, we were able to measure the impact a 'term' (i.e., P, I or D) is having in the equation. In other words, the less the 'term' value the less significant role a 'term' is playing in specifying the next sampling fraction. For instance, in the second values setting, we set the term D to the value 0.2, indicating that we want to avoid playing with the stability of the system by only moderately (as small as 0.2) accounting for a future prediction of the data oscillation. In other words, we account for the trends of future data load slowly. We apply the same for SAOS and SRS-based sampling. In addition, we emulate the oscillating trend of data arrival rates (i.e., batch size) in real scenarios by alternating rates with the following values. '500K to 2000K', '500K to 3000K', '500K to 5000K', '500K to 2000K to 1000K'. In doing so, we can measure the ability of SpatialSSJP in responding to sudden oscillations in data arrival rates. We compare our design with a plain spatial join operator without sampling. 2) *SpatialSSJP ability to satisfy accuracy target by applying the accuracy-aware rate controller*. We fix the arrival rate and change the accuracy target (expressed as a 'margin of error' value) between some stringent value that equals to 0.01, a 'middle strictness' value that is equal to 0.03 and a permissive value of 0.09. We compare the join operator using SAOS against the join operator using an SRS-based counterpart.

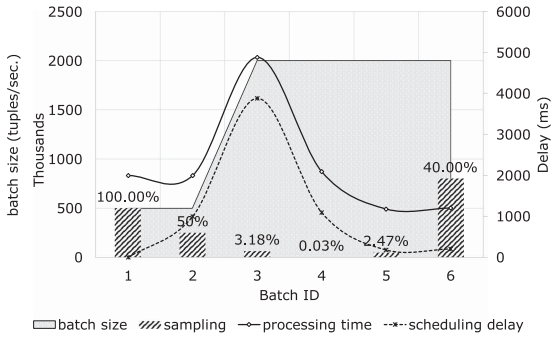

Fig. 3. Catch up at PID values 11,1 where SpatialSSJP can meet the latency target by applying the 'latency-aware controller' for reducing the join input tuples using SAOS where the oscillation is '500k to 2000k'. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction at each 'batch interval'.

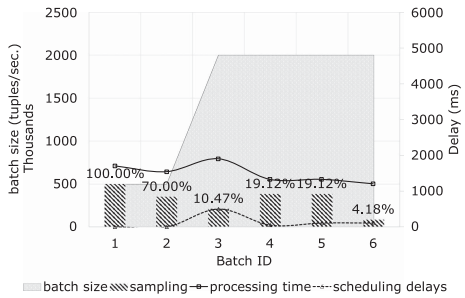

Fig. 4. Catch up at PID values that are equal to (10.60.2), using SAOS as a pre-join sampling method, oscillation is '500k to 3000k.

B. Results Discussion

All results reported in this paper are calculated as the median (i.e., 50th percentile) of running the system for ten times.

1) The Effect of the Pre-Join Sampling Design on the Ability of SpatialSSJP to Meet Latency Goal: We compare the effect of applying various pre-join sampling designs on the ability of the SpatialSSJP latency controller in catching up with the oscillation in data arrival rates. Also, we compare our system to a plain Spark join version where no pre-join sampling is performed.

Fig. 3 shows the ability of our latency-aware controller (part of SpatialSSJP) in lowering the latency to the minimum (close to zero). We have applied the PID values that are equal to (11,1), respectively in this case. Scheduling and processing delays (processing delay is computed as the difference between the processing time and the batch interval, which happens to be equal to '1' second in all experiments that we have performed) converge initially at a confluence point, where our latency-aware controller senses that a probable staggered delay is blamed to a sudden spike in the data batch size (roughly from 500K to 2000k), thereafter a newly calculated sampling rate that is roughly equals to 0.03% is served to SAOS pre-join sampler in the front-stage. A catch up then occurs. Because the oscillation is relatively high (from 500K to 2000K), the system reaches batch interval number '5' without returning to normal operation, meaning that the accumulated number of extra records is appropriately considered. Also, the fact that we have set the term 'D' to a value that equals '1' to signify that we greatly account for another future approaching spike, which explains then the low value of the sampling fraction in batch intervals '4' and '5'.

Notice that since we rely on tumbling window semantics (i.e., non-overlapping time-based windows), processing time never comes below the duration of 'batch interval'.

Fig. 4 shows the effect of changing the PID values to (10.60.2) respectively (scientifically plausible values). Notice the effect of a lower 'D' term value. We do not account excessively for a future spike in the data arrival rates. Hence, the sampling fractions are more permissive.

Similar trends occur for both PID values combinations ('11,1,' and '10.60.2') in cases of less aggressive spikes in the arrival rates. We have specifically tested for a spike from 500k to 2000K tuples/second.

On the contrary, for testing SpatialSSJP under more stringent arrival rates, we have tested specifically with a sudden spike from 500K to 5000K. SpatialSSJP was able to survive that spike and catch up occurred for both PID combinations ('11,1' and '10.60.2').

To further exhibit the elasticity of SpatialSSJP, we have emulated a fluctuation that spikes suddenly between 500K to

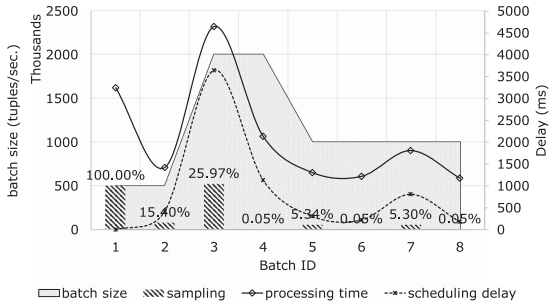

Fig. 5. Catch up at PID values (10.60.2) and oscillation '500k-2000K-1000K' using SAOS as a pre-join sampling method.

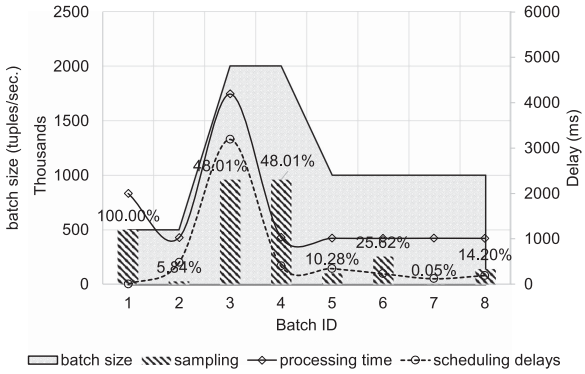

Fig. 6. Catch up (SRS) at PID values (10.60.2) and oscillation '500k-2000K-1000K' using SRS as a pre-join sampling method.

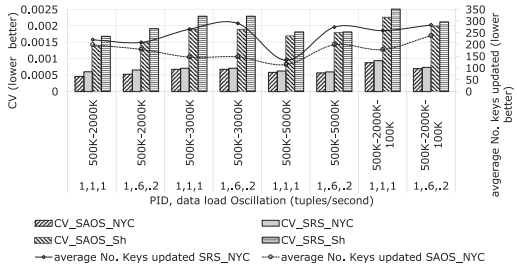

Fig. 7. Coefficient of Variance (CV) by applying SAOS against SRS-based baseline, both pre-join sampling methods, feeding samples to the stream-static join operator of SpatialSSJP. Varying PID values between (11,1) and (10.60.2). 'sh' is for 'Shenzhen' and 'NYC' for NY City.

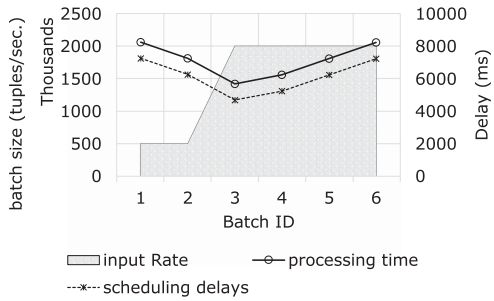

Fig. 8. High delays imposed by disabling sampling during burst loads, Oscillation 500K – 2000K. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main access to the left shows the batch size (input rate).

2000K, then slows down to 1000K. Fig. 5 shows the elasticity of SpatialSSJP, notice the 'similar discernible pattern' as the input rate that the times (scheduling and processing) are following. Our method can extrapolate unseen sudden spikes in data arrival rates.

For all settings, processing times fall as a convergence occurs in such a way that they fall inside the boundaries of batch intervals (1000 milliseconds in this case). After that stable point is reached, SpatialSSJP starts to pull more samples in each batch interval.

As a way of contrast, by employing an SRS-based baseline pre-join sampling method instead of our SAOS sampling method, SpatialSSJP was also able to remain alive at all spikes. However, it hits more Standard Errors (SE) and CVs than SAOS for same PID combinations, suggesting that the pre-join sampling design has a property that affects accuracy goals. Fig. 6 illustrates an example.

Despite that both pre-join sampling methods SAOS and the baseline that is based on SRS can cause the stream-static join operator to survive spikes in streaming data loads under SpatialSSJP, SAOS is preferred against the SRS-based baseline as a pre-join sampling design. This is attributed to the fact that SAOS yields better sampling geo-statistics for estimating target variables over stream-static join outputs as opposed to an SRS-based baseline counterpart. The CV values of Fig. 7 (for both datasets, NY City and Shenzhen) show the trend for PID values that are equal to (11,1) respectively. Similar trend occurs for the other combination of PID values (10.60.2).

Disabling sampling in fluctuating streaming environments results in a detrimental effect that causes the DSP system to become unavailable during spikes in the arrival rates due to the computationally expensive stream-static geospatial join. We have tested for an oscillation case of '500K to 2000K' as shown in Fig. 8. For a more stringent spike (500K to 5000K), applying no pre-join sampling results in a fatal situation where the system throws an out-of-memory (OOM) exception. This constitutes the baseline version of SpSS geospatial stream-static join without a pre-join sampling and/or a latency controller. Hence, applying qualified pre-join sampling with feedback-based auto-recovery in geospatial approximations over joins in data streaming highly dynamic application scenarios is essential for the DSP system to survive sudden spikes in data arrival rates, while preserving the geo-statistical properties of the join output.

Results we have shown in this subsection refer to NY City datasets for measuring the effect of the pre-join sampling design on the ability of SpatialSSJP to meet latency goal. We have tested with the second dataset (Shenzhen dataset), and test results show the same discernible pattern as that of NY City, despite not shown in the paper for the lack of space. With Shenzhen data, our system can meet the latency requirements for all configurations that have been discussed in this experimental section (i.e., the "oscillating trend of data arrival rates" and "PID value settings"), showing

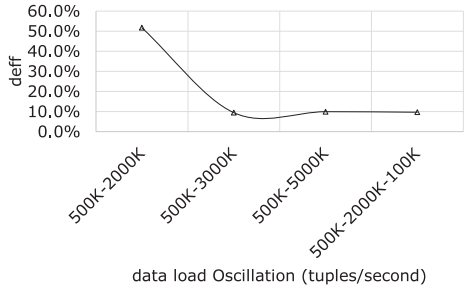

Fig. 9. Design effect of SAOS as a pre-join sampling methods, feeding samples to the stream static join operator of SpatialSSJP instead of the SRS-based baseline counterpart. PID values are 10.60.2.

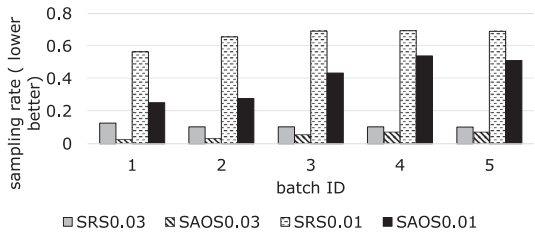

Fig. 10. Accuracy gain by applying SAOS with SpatialSSJP against SRS baseline. In the legend, 'moe0.03' means 'margin of error' that equals 0.03, whereas 'moe0.01' means 'margin of error' that equals 0.01.

discernible patterns that are similar to what appear in Figs. 3, 4, 5, and 6.

We conclude that by relying on SAOS as a pre-join sampling method instead of an SRS-based counterpart, we achieve a better gain (expressed as a 'design effect' or 'deff' for short) as shown in Fig. 9. This applies to PID values that are equal to (1, 0.6, 0.2). Similar trend occurs for the other PID values combination. The figure suggests that by applying SAOS as a pre-join method instead of SRS, we achieve a better performance, effectively outperforming SRS by a significant margin. On average, the relative achievement over SRS is at least 10% and tipping up to 50%, at times, to suggest that relying on a stream-static pre-join sampling method (such as SAOS) that is aware of the data shape (being spatial or geo-referenced) should be preferable over random designs that are not aware of the spatial characteristics of data.

2) The Effect of the Pre-Join Sampling Design on SpatialSSJP Ability to Satisfy Accuracy Target: To test the ability of SpatialSSJP in achieving a prespecified accuracy target (expressed as a 'margin of error' value) for geo-statistical aggregation approximations over geospatial stream-static joins, we have compared the effect of the pre-join sampling design on the approximation accuracy. Particularly, we compare relying on SAOS as a pre-join sampling method against the SRS-based baseline counterpart. As shown in Fig. 10, on average, a stream-static join requires sampling fractions when SAOS is the pre-join sampling method, which are less than those required by the SRS-based baseline counterpart in order to achieve the same accuracy target of the approximation result over the join output. This applies for all 'margin of error' values (for example, 0.03 and 0.01 as shown in Fig. 10). As it is clear from Fig. 10, in stringent cases (where the target 'margin of error' equals 0.01), the approximator over the geospatial join requires more samples, regardless of which among the pre-join sampling, SAOS or SRS, is applied. Be that as it may, the approximator always requires less sampling fractions by relying on SAOS as a pre-join sampling method as compared to the SRS-based counterpart to achieve the prespecified accuracy target of the approximation over the join output.

Referring to (6) that calculates the number of samples by applying SRS, since N (continuous population at every batch) is relatively large, n0/N becomes tiny, yielding $n \approx n0$. As such, mostly the same sample size is required for any large population (be that as it may, one million or even one billion tuples). This explains the reason behind obtaining almost the same sampling fraction for subsequent batch intervals (whether applying SAOS or SRS, the same discernible pattern applies), which corroborates the formalization herein.

VI. Related Works

Spatial join is a prohibitively expensive operation which can render most advanced SPE systems irresponsive during spikes in data loads [1]. To alleviate this problem, various works in the relevant literature have designed frameworks for controlling the arrival rates of data coming from streaming sources.

A. Stream-Stream Join Processing

Most of the methods in the literature concentrate on either stream-stream [12] or static-static joins for distributed environments. However, the focus on stream-static join processing is still largely unexplored [20]. Stream-stream differs from the stream-static join in that data for both sides of the join in the former is coming from streaming sources. Also, both sides of the static-static join are static relations with pre-known data. This is so because stream-stream join runs into challenges that are not normally present in a stream-static counterpart. To clarify, at any point of time, the view of both sides of the stream-stream join (the two data streams) is incomplete, which makes it challenging to find matches between the two data streams. Any tuple from one of the streams can potentially match with any upcoming (have-not-yet-arrived) tuple from the other stream. This has an implication on making it harder to decide which tuples to drop from each individual data stream, even knowing the sampling rates. Having said that, we leave the tailoring of our system so that it supports stream-stream scenarios as a future research perspective.

B. Distributed Stream-Static Spatial Join Processing

Filter-and-refinement approach is an essential technique for spatial join processing [21]. It first performs MBR-join to return candidate tuples, then it checks whether those objects satisfy the spatial join predicate. In addition to Spark's Magellan, several other distributed geospatial processing frameworks employ the filter-and-refinement approach for efficient spatial query processing (including the spatial join processing). For example,

Apache Sedona (previously GeoSpark [21]) and LocationSpark [22]. Simba [23] is a similar system atop Spark that does not utilize the filter-and-refinement model for spatial join. However, systems like GeoSpark and LocationSpark utilize tree-based indexing structures, requiring thus, 10%-40% additional space for storing the tree node information [24]. In addition, Hadoop-based systems such as SpatialHadoop [25] store intermediate results on disk rather than in-memory, resulting then in less memory utilization as opposed to main-memory frameworks [24]. What's more, despite the abundance of works in literature comparing distributed systems for spatial processing [26], there is no work that compares the performance of those systems in data stream settings, to the best of our knowledge.

Indexing data streams on-the-fly constitutes an integral part of online spatial join. Spatial join techniques can be classified into two categories: internal memory methods (e.g., z-order curves), or external memory methods (e.g., tree-based structures that require hierarchical traversal). Since one side of the join is a fast-arriving data stream, and because we choose to operate with micro-batch processing mode, then data streams every batch interval fit into main-memory. This encouraged us to choose a join method that is based on filter-and-refinement and z-order curves (internal memory method for spatial join) as they perform better than external memory structures (hierarchical tree-based structures such as R-Trees) on modern parallel hardware [8].

Having said that, we have selected Spark's Magellan as a representative code base library to prototype our SpatialSSJP novel system because spatial join in stream-static join is performed in main-memory, given the fact that we are operating on a micro-batch mode. Thus, an internal memory technique such as z-order curves is preferred over counterparts at this stage. Comparing the performance of stream-static join between Spark's Magellan and other similar frameworks seems interesting, but we consider it outside the scope of this paper, and we leave it as a future research perspective. We recapitulate also that the focus of this paper is the incorporation of geospatial approximate query processing mechanisms with a pipeline containing a stream-static join operator. Thus, we argue that our framework is seamlessly transferable to plenty of in-memory distributed spatial data stream processing frameworks, including Apache Sedona (previously GeoSpark [27]) and GeoMesa [28].

A significant recent work in static-static join appears in [29], where the authors have focused on improving spatial join algorithms that involve both relations as static tables (one that contains spatial points, while the other contains polygons). They basically have designed a novel multi-level on-the-fly spatial indexing scheme that has proven efficient in terms of time-based QoS goals.

C. Approximate Stream-Static Spatial Join Processing

From the approximation side, [30] applies load shedding to stream-static join. They employ a straightforward formula for computing the latest batch size, where they shed extra loads if an amount that is roughly equals to double of a threshold value is exceeded to prevent accumulating tuples in the buffer. Two problems are apparent in this design. First, it is not attuned to the data shapes (the fact that most arriving streaming data is geo-referenced) because it is shedding data randomly. Second, the continuous cycle of extra load spilling to (when load exceeds capacity) and recovering from (when loads slow down) disks imposes an extra cost that increases the I/O overhead incurred.

As far as we know, only a few studies have given attention to the micro-batch semantics for optimizing approximate stream-static joins. As an example, [20] presents a framework termed DS-join for the join between streaming sources and data-at-rest (static) using the micro-batch semantics of recent distributed SPEs. Authors basically focus on distributing join execution in cases where the static table does not comfortably fit in-memory of the Spark worker nodes.

Most distributed geospatial DSP systems, SpatialHadoop [25], HadoopGIS [31] and SpatialSpark [32], are batch-oriented, which are not designed to work with streams.

To the best of our knowledge, there are no works in literature that specifically focus on approximate processing (by relying basically on sampling) in supporting stream-static joins for spatial workloads. However, few works have applied sampling for general data streaming workloads in fluctuating dynamic application settings. For instance, [33] proposes a framework titled AccStream as an adaptive overload management system (residing atop Spark Streaming [11]) which samples data tuples from a general data source. AccStream consists of three components: a controller, collector and a receiver that is repurposed and retrofitted from that of the Spark's stock version. The job of the collector is to feed statistical information (such as latency and accuracy, where accuracy depends on sampling theory) to the controller. Thereafter, the controller calculates an appropriate sampling rate. The receiver component is a repurposed and retrofitted version of the plain receiver in Spark Streaming. Aiming at achieving the latency goals, they employ a dynamic and self-tuning learning-based model (i.e., latency model). The disadvantage, however, is that AccStream is a general-purpose framework that is not specifically designed to work with spatial data loads. In addition, the system employs a computationally expensive method for predicting future spikes, which negatively affects the system's long-term stability. This is attributed to the fact that the method requires computing many statistics that are not readily accessible through the underlying system codebase, causing an overhead to accumulate and carry over to subsequent time windows.

We are not aware of any system from the relevant literature that accomplishes the goals we achieved by designing SpatialSSJP.

VII. Conclusion and Future Work

In this paper, we designed and implemented a novel system for adaptive approximate query processing over geospatial stream-static join operations. Our system can achieve and plausibly tradeoff prespecified time-based and estimation-quality QoS goals. We prototyped our system (that we call SpatialSSJP) atop Spark Structured Streaming and assessed it against state-of-art baselines, using big real-world mobility data. Our results conclude that SpatialSSJP achieves latency and accuracy goals

defined in the user query requirement as compared to baseline systems.

Future works include porting the sampling methods to Edge devices near the data sources, in addition to applying other prefiltering methods prior to the sampling, aiming to reduce the join data shuffled across the network. Also, the current version of SAOS selects the same percentage of tuples from each region in the embedding space. An interesting future research perspective is to consider the number of vertices in each region as a configurable parameter for specifying the percentage for each region independently. This way, we can choose to select higher percentage of tuples from areas with less vertices as they require cheaper point-in-polygon tests as opposed to regions with much higher number of vertices.

REFERENCES

[1] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Efficiently integrating mobility and environment data for climate change analytics," in *Proc. IEEE 26th Int. Workshop Comput. Aided Model. Des. Commun. Links Netw.*, 2021, pp. 1–5.

[2] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "QoS-aware approximate query processing for smart cities spatial data streams," *Sensors*, vol. 21, no. 12, 2021, Art. no. 4160.

[3] I. M. Al Jawarneh, P. Bellavista, L. Foschini, and R. Montanari, "Spatial-aware approximate Big Data stream processing," in *Proc. IEEE Glob. Commun. Conf.*, 2019, pp. 1–6.

[4] M. Armbrust et al., "Structured streaming: A declarative api for real-time applications in apache spark," in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, 2018, pp. 601–613.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 95.

[6] M. Armbrust et al., "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.

[7] R. T. Whitman, M. B. Park, B. G. Marsh, and E. G. Hoel, "Spatio-temporal join on apache spark," in *Proc. 25th ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2017, pp. 1–10.

[8] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, pp. 7–es, 2007.

[9] A. Arasu et al., "Stream: The stanford data stream management system," in *Data Stream Management*. Berlin, Germany: Springer, 2016, pp. 317–336.

[10] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Spatially representative online Big Data sampling for smart cities," in *Proc. IEEE 25th Int. Workshop Comput. Aided Model. Des. Commun. Links Netw.*, 2020, pp. 1–6.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. Twenty-4th ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.

[12] A. Shahvarani and H.-A. Jacobsen, "Distributed stream KNN join," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 1597–1609.

[13] X. Chen, Y. Vigfusson, D. M. Blough, F. Zheng, K.-L. Wu, and L. Hu, "GOVERNOR: Smoother stream processing through smarter backpressure," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017, pp. 145–154.

[14] S. L. Lohr, *Sampling: Design and Analysis*. Toronto, ON, Canada: Nelson Education, 2009.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1997, pp. 171–182.

[16] P. Carbone, A. Katsifodimos, and S. Haridi, *Stream Window Aggregation Semantics and Optimization*, Berlin, Germany: Springer, 2019.

[17] *New York, NY, USA (N.Y.). Taxi and Limousine Commission. New York, NY, USA City Taxi Trip Data*, 2019-02-20 ed. Ann Arbor, MI, USA: Inter-Univ. Consortium for Political and Social Research [distributor], 2009-2018.

[18] G. Wang, X. Chen, F. Zhang, Y. Wang, and D. Zhang, "Experience: Understanding long-term evolving patterns of shared electric vehicle networks," in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, 2019, pp. 1–12.

[19] R. Derakhshan, A. Sattar, and B. Stantic, "A new operator for efficient stream-relation join processing in data streaming engines," in *Proc. 22nd ACM Int. Conf. Inf. Knowl. Manage.*, 2013, pp. 793–798.

[20] Y.-H. Jeon, K.-H. Lee, and H.-J. Kim, "Distributed join processing between streaming and stored Big Data under the micro-batch model," *IEEE Access*, vol. 7, pp. 34583–34598, 2019.

[21] J. Yu, Z. Zhang, and M. Sarwat, "GeoSparkViz: A scalable geospatial data visualization framework in the apache spark ecosystem," in *Proc. 30th Int. Conf. Sci. Stat. Database Manage.*, Bozen-Bolzano, Italy, 2018, Art. no. 15, doi: 10.1145/3221269.3223040.

[22] M. Tang, Y. Yu, A. R. Mahmood, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: In-memory distributed spatial query processing and optimization," *Front. Big Data*, vol. 3, 2020, Art. no. 30.

[23] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1071–1085.

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

[25] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1352–1363.

[26] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1661–1673, 2018.

[27] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proc. 23rd SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2015, pp. 1–4.

[28] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "Geomesa: A distributed architecture for spatio-temporal fusion," in *Geospatial Informatics, Fusion, and Motion Video Analytics V*, Bellingham, WA, USA: SPIE, 2015, pp. 128–140.

[29] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Efficient QoS-aware spatial join processing for scalable NoSQL storage frameworks," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 2, pp. 2437–2449, Jun. 2021.

[30] M. A. Naeem, G. Dobbie, C. Lutteroth, and G. Weber, "Skewed distributions in semi-stream joins: How much can caching help?," *Inf. Syst.*, vol. 64, pp. 63–74, 2017.

[31] A. Aji et al., "Hadoop-GIS: A high performance spatial data warehousing system over MapReduce," in *Proc. VLDB Endowment Int. Conf. Very Large Data Bases*, 2013, pp. 1009–1020.

[32] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *Proc. IEEE 31st Int. Conf. Data Eng. Workshops*, 2015, pp. 34–41.

[33] H. Sun, R. Birke, W. Binder, M. Björkqvist, and L. Y. Chen, "AccStream: Accuracy-aware overload management for stream processing systems," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017, pp. 39–48.

Isam Mashhour Al Jawarneh (Member, IEEE) received the PhD degree in computer science and engineering from the University of Bologna, Italy, in 2020. Since 2020, he had been a postdoctoral research fellow with the University of Bologna. His research mainly focuses on spatial data science, in addition to the design and development of novel Cloud and Edge-based Big Data management methods, aiming at synergistically advancing geospatial, time-series, and contextual aspects.

Paolo Bellavista (Senior Member, IEEE) received MSc and PhD degrees in computer science engineering from the University of Bologna, Italy, where he is now a full professor with distributed and mobile systems. His research interests include activities span from pervasive wireless computing to location/context-aware services, from edge cloud computing to middleware for Industry 4.0 applications. He is currently the scientific coordinator of a large H2020 Big Data innovation action called IoTwins about distributed digital twins for the manufacturing industry. He serves on the Editorial Boards of *IEEE Communications Surveys and Tutorials*, *ACM Computing Surveys*, *IEEE Transactions on Network and Service Management*, *Elsevier Pervasive Mobile Computing*, and *Elsevier Journal on Network and Computing Applications*, among the others.

Antonio Corradi (Senior Member, IEEE) received the graduate degree from the University of Bologna, Italy, and the MS degree in electrical engineering from Cornell University, USA. He is a full professor of computer engineering with the University of Bologna. His research interests include distributed systems, middleware for pervasive and heterogeneous computing, infrastructure for services, and network management.

Luca Foschini (Senior Member, IEEE) received the graduate degree from the University of Bologna, Italy, and the PhD degree in computer science engineering, in 2007. He is now an associate professor of computer engineering with the University of Bologna. His research interests include span from integrated management of distributed systems and services to wireless pervasive computing and scalable context data distribution infrastructures and context-aware services. Currently, he is working on mobile crowdsensing and crowdsourcing and management of Cloud systems for Smart City environments.

Rebecca Montanari (Member, IEEE) received the graduate degree from the University of Bologna, and the PhD degree in computer science engineering in 2001. She is now a full professor of computer engineering with the University of Bologna. Her research interests include primarily focuses on semantic-based middleware supports for service provisioning, context-aware services, security solutions for pervasive environments, policy-based service management, and adaptive and scalable middleware solutions for system and service management.