Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA (Ph.D.) IN

Computer Science and Engineering

Ciclo XXXII

Settore Concorsuale: 09/H1

Settore Scientifico Disciplinare: ING-INF/05

Quality of Service Aware Data Stream Processing for Highly Dynamic and Scalable Applications

Presentata da:

Isam Mashhour Hasan Al Jawarneh

Coordinatore Dottorato:

Prof. Davide Sangiorgi

Supervisore:

Prof. Rebecca Montanari

Esame finale anno 2020

website: https://www.unibo.it/sitoweb/isam.aljawarneh3

E-mail: isam.aljawarneh3@unibo.it

Abstract

Huge amounts of georeferenced data streams are arriving daily to data stream management systems that are deployed for serving highly scalable and dynamic applications. There are innumerable ways at which those loads can be exploited to gain deep insights in various domains. Decision makers require an interactive visualization of such data in the form of maps and dashboards for decision making and strategic planning. Data streams normally exhibit fluctuation and oscillation in arrival rates and skewness. Those are the two predominant factors that greatly impact the overall quality of service. This requires data stream management systems to be attuned to those factors in addition to the spatial shape of the data that may exaggerate the negative impact of those factors. Current systems do not natively support services with quality guarantees for dynamic scenarios, leaving the handling of those logistics to the user which is challenging and cumbersome. Three workloads are predominant for any data stream, batch processing, scalable storage and stream processing. In this thesis, we have designed a quality of service aware system, SpatialDSMS, that constitutes several subsystems that are covering those loads and any mixed load that results from intermixing them. Most importantly, we natively have incorporated quality of service optimizations for processing avalanches of geo-referenced data streams in highly dynamic application scenarios. This has been achieved transparently on top of the codebases of emerging de facto standard best-in-class representatives, thus relieving the overburdened shoulders of the users in the presentation layer from having to reason about those services. Instead, users express their queries with quality goals and our system optimizers compiles that down into query plans with an embedded quality guarantee and leaves logistic handling to the underlying layers. We have developed standard compliant prototypes for all the subsystems that constitutes SpatialDSMS. Thereafter, we have tested with huge amounts of real and synthetic geo-referenced datasets, deploying our computing clusters in-house and in Cloud computing environments. Our results show that all the subsystems of SpatialDSMS were able to achieve the envisaged quality goals and outperform baselines by significant margins.

Lovingly, to my parents, my son and my wife

Ai miei genitori, a mio figlio e a mia moglie,

per il loro instancabile sostegno.

Acknowledgements

I first would like to sincerely thank my Ph.D. supervisor Prof. Rebecca Montanari for magnificently and wisely supervising my works during my three years Ph.D. I am thankful and grateful to Prof. Paolo Bellavista for encouraging me and inviting me to his fabulous research group. Many thanks to my Ph.D. advisor Prof. Antonio Corradi for advising me and supporting my research. I would like to thank Prof. Luca Foschini for his patience and for motivating me in every aspect while he was helping me to improve my research skills and competences. Thanks to you all, with your supervision, support, guidance and advices, I have enjoyed every moment during my Ph.D. journey. Thanks a lot for making me always feel like being with family, for being my friends and always encouraging me and supporting me with deep tenderness and warmness.

I am very grateful to my family, my parents, my son and my wife, my sisters and my brothers for the love, support and encouragement they give to me.

I also would like to thank my collaborators from the department of computer science and engineering (DISI) at University of Bologna (Andrea Zanotti and Francesco Casimiro) for their technical contributions. I would like to thank my fellow doctoral researchers and research assistants, Domenico Scotece, Michele Solimando, Riccardo Venanzi, Giuseppe Martuscelli, and all the other fellows in Lab2 for being my friends and for the nice work environment we have shared.

I also would like to thank the external reviewers of my Ph.D. thesis, Prof. Melike Erol-Kantarci and Prof. Javier Berrocal, for their insightful comments, feedback and suggestions that assist in improving the quality of this work.

I am thankful to the Institute of Advanced Studies (ISA) at the University of Bologna for granting me the PhD@ISA fellowship for three years. I am specifically thankful to the director of ISA, Prof. Dario Braga, thank you for the great arrangement of those insightful and fruitful seminars. I have learned a lot during those seminars.

A very special gratitude goes to Centro Interdipartimentale di Ricerca Industriale su ICT (CIRI ICT) at University of Bologna for funding my works for the three years during my Ph.D. I have enjoyed a lot working on SACHER project.

Publications

Part of the contents of this thesis are based on the following publications:

[1] **I. M. Al Jawarneh**, P. Bellavista, L. Foschini and R. Montanari, "Spatial-aware approximate big data stream processing," in 2019 IEEE Global Communications Conference, GLOBECOM. In press.

[2] **I. M. Al Jawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari and A. Zanotti, "In-memory spatial-aware framework for processing proximity-alike queries in big spatial data," in 2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2018, pp. 1-6.

[3] **I. M. Al Jawarneh**, P. Bellavista, F. Casimiro, A. Corradi and L. Foschini, "Cost-effective strategies for provisioning NoSQL storage services in support for industry 4.0," in 2018 IEEE Symposium on Computers and Communications (ISCC), 2018, pp. 1227.

[4] **I. M. Aljawarneh**, P. Bellavista, C. R. De Rolt and L. Foschini, "Dynamic identification of participatory mobile health communities," in Cloud Infrastructures, Services, and IoT Systems for Smart Cities. Springer, 2017, pp. 208-217.

[5] **I. M. Aljawarneh**, P. Bellavista, A. Corradi, R. Montanari, L. Foschini and A. Zanotti, "Efficient spark-based framework for big geospatial data query processing and analysis," in 2017 IEEE Symposium on Computers and Communications (ISCC), 2017, pp. 851-856.

[6] **I. M. Aljawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari , "Efficient QoS-Aware Spatial Join Processing for NoSQL Scalable Storage Frameworks". 2020. **Submitted**.

[7] **I. M. Aljawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari, ". SpatialSSJP: QoS-Aware Adaptive Approximate Stream-Static Spatial Join Processor". 2020. **Submitted**.

[8] **I. M. Aljawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari, " Locality-Preserving Spatial Partitioning Scheme for Quality Spatial Analytics in Distributed Main Memory Frameworks".2020. **Submitted**.

[9] **I. M. Aljawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari. "QoS-Aware Optimizations for Big Geospatial Data Management - A Survey". 2020. **Submitted.**

Additional publications published while at the department of computer science and engineering, University of Bologna during my Ph.D.:

[10] **I. M. Al Jawarneh**, P. Bellavista, A. Corradi, L. Foschini, R. Montanari, J. Berrocal and J.M. Murillo. “A Pre-filtering Approach for Incorporating Contextual Information into Deep Learning Based Recommender Systems”. 2020. IEEE Access. To appear.

[11] S. Bertacchi, **I. M. Al Jawarneh**, F. I. Apollonio, G. Bertacchi, M. Cancilla, L. Foschini, C. Grana, G. Martuscelli and R. Montanari, "SACHER project: A cloud platform and integrated services for cultural heritage and for restoration," in Proceedings of the 4th EAI International Conference on Smart Objects and Technologies for Social Good, 2018, pp. 283-288.

[12] **I. M. Al Jawarneh**, P. Bellavista, L. Foschini, R. Montanari, J. Berrocal and J. M. Murillo, "Toward privacy-aware healthcare data fusion systems," in International Workshop on Gerontechnology, 2018, pp. 26-37.

[13] **I. M. Al Jawarneh**, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in ICC 2019-2019 IEEE International Conference on Communications (ICC), 2019, pp. 1-6.

[14] **I. M. Al Jawarneh**, P. Bellavista, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli and F. Bosi, "Qos and performance metrics for container-based virtualization in cloud environments," in Proceedings of the 20th International Conference on Distributed Computing and Networking, 2019, pp. 178-182.

Contents

Chapter 1

Introduction

The unprecedented abundance of Internet of Things (IoT) devices have caused avalanches of ultra-fast arriving geo-referenced data streams to arrive at Data Stream Management Systems (DSMS). Analyzing that data is important for strategic planning and decision making. Processing such massive amounts of data in a timely fashion depending on relational systems is specifically grueling. Consequently, novel frameworks have emerged, such as Apache Spark [1] ,for distributed processing, and MongoDB [2] for scalable distributed storage. Those general-purpose systems have established themselves as de-facto standards for macro-scale big data intensive analytics. However, employing them as-is in dynamic and highly scalable application scenarios (e.g., smart cities [3] , Industrial Internet of Things (IIoT), Industry 4.0 [4] and urban computing [5]) requires investigating their ability in meeting QoS goals (e.g. latency/throughput and resource utilization) . An intrinsic problem in those frameworks is that they are not natively attuned to data characteristics, rendering them unable to achieve (or at least striking a plausible balance between) QoS goals. Those systems do not provide out-of-the-box representational and analytical models for geospatial data, overburdening developers with logistics and slowing down the production process.

Dynamic and highly scalable application scenarios coming from smart cities, IIoT and urban computing are innumerable. However, they all require mixing (sometimes in a mashup fashion, thus fusing disparate elements) workloads in order to get full insights that guide the decision making for improving our lives in all aspects. Batch processing and online aggregate processing occupy a big share of those workloads. The fact that input data is mostly geo-referenced invokes a novel spatial aware data stream management system that covers most critical quality of service aspects in an end-to-end pattern.

Current big data management systems (for example, Apache Spark [1] and MongoDB [2]) are growing quickly. Having modular architectures, those systems, despite not being attuned to characteristics of georeferenced data, are promising jumping-off points that can be used for building optimized QoS aware versions, achieving goals related to latency/throughput, accuracy and resource utilization. In this thesis, we generally aim at improving the QoS of

those mature systems for highly dynamic application scenarios. In the next subsection, we introduce a genuine QoS-demanding highly dynamic application scenario that motivates various contributions that we have achieved in this thesis.

This chapter is organized as follows. In § 1.1, we iterate a contrived representative toy scenario in highly dynamic application domain. In what follows, we introduce the thesis statement in § 1.2. Thereafter, in § 1.3, we summarize our contributions in this thesis in a coherent and consistent structure. We conclude the chapter in § 1.4 by presenting an outline showing the organization of remaining parts of the thesis.

1.1 Highly Dynamic and Scalable Applications: A Motivating Scenario and Usage Model

Our main goal by explaining the following highly scalable and dynamic application scenario is to delineate, in a coherent and consistent manner, the contributions presented in this thesis.

Streaming data coming from heterogeneous sources in dynamic applications, such as smart cities, can be exploited in innumerable ways to get deep insights that improve the quality of our lives. In this section, we envision a representative application scenario that belongs to the family of participatory healthcare, a life-critical dynamic scenario that imposes harsh QoS goals on the underlying data stream management system (DSMS). QoS goals may include low-latency, high-throughput, high-accuracy and high resource utilization.

Consider an application which analyzes Global Positioning System (GPS) data collected in real-time by citizens and vehicles moving around in a city. A citizen suffering a chronic disease (e.g., asthma attack) which may attack suddenly while moving around in a city and needs an instant first-aid. The goal is to provide reliable assistance to that patient and keep danger as low as possible, while at same time avoid disorganizing roads traffic (i.e., avoid causing congestions). Achieving those goals requires two things. First, sending patient location and health severity degree to the nearest hospital. Second, finding nearest appropriate person who is willing and able (well-trained) to provide first-aid. This is a mixed-workload scenario which invocates a *reliable* system that needs to provide at least the following services:

1) **Traffic Light Controller (TLC)**. This component constitutes sensors that are implanted in the roads and can send timely signals to a periodic traffic signal actuator. Actuator then decides to change lights of some traffic lights into green for those lanes to allow the ambulance to pass in a consistent way.
2) **Smart Real-time Pathfinder (SRP).** This component of the envisaged system generates an interactive navigation map that navigates ambulances en-route to accident location, whereas recommends alternative roads to other vehicles consistently.
3) **(Near) Real-time Community Detection (RCD).** This component can identify communities in the surroundings of the patient by applying a clustering and selects the most appropriate volunteer who is the nearest and capable of providing first-aid.

A typical architecture of a system that handles this scenario can be envisioned in the schematic diagram of figure 1.1, showing a typical interplay and interaction between many constituting components. This resembles a publish/subscribe pattern, where data collectors dynamically send geo-referenced data to batch and online processing systems that perform analytics and serve them to subscribers (e.g., actuators) that enact/react correspondingly.

Various QoS goals are imposed at all stages during the interaction of the constituent parts of the envisaged DSMS shown in Figure 1.1. TLC should act in a *latency* bounded fashion (i.e., low-latency QoS goal) to control traffic light signals efficiently in real-time. Errors are not allowed in such a critical service and nearly-perfect *accuracy* is a must. On the contrary, SRP

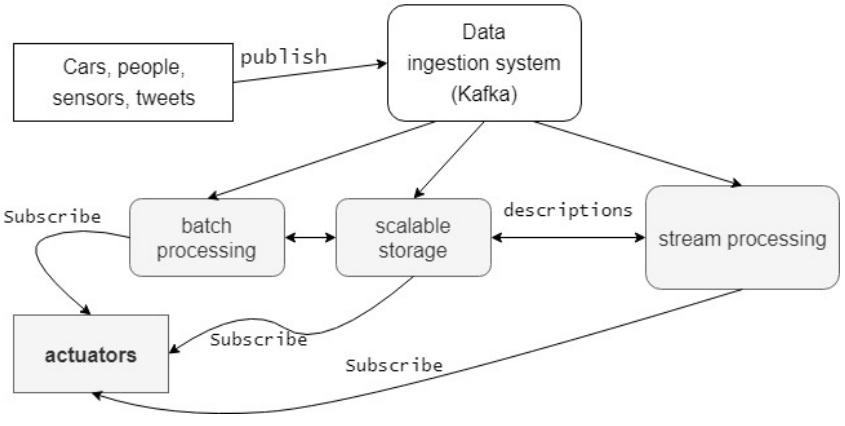

Figure 1.1. A typical publish/subscribe based pattern showing the interaction between typical system components in a typical highly dynamic and scalable application scenario

can safely depend on approximation, thus trading off a rigorous error-bounded *accuracy* for lower *latency*. This in part is because we need to generate an approximate heatmaps in real-time to draw trajectories for ambulances en-route to accident locations, only showing approximately how congested a lane in a specific time is enough. Also, RCD can be adequately based on an error-bounded approximate (i.e., high estimation accuracy QoS goal).

A comprehensive usage model could extend tremendously, but even with this simple synopsis it is quite straightforward to distinguish an important aspect that is inherently apparent in this scenario, that it requires mixing *instantaneous reactions* with *proactive actions*. Reactions include applying real-time analytics before data becomes obsolete and loses its value. Proactive actions require achieving data in a homogeneous way so as to apply predictive models (e.g., overnight) that further assist in decision making and planning. To be considered a QoS-aware DSMS for highly dynamic application scenarios, it should incorporate QoS-awareness natively within the layers of its core baselines without requiring developers and users to reason about the underlying mechanisms of those services.

1.2 Thesis Statement

In this thesis, we aim at designing and implementing a constellation of methods and algorithms, then incorporating them into few sub-systems that collectively form a spatial data stream management system for managing and processing spatial streaming data in highly dynamic and scalable applications. Low latency, high throughput, and controlled accuracy with rigorous error-bounds, in addition to high resources utilization are QoS goals of a paramount importance. We aim at trading them off appropriately in a manner that improves the overall service quality envisaged from the system.

1.3 Thesis Contributions

In this thesis, we show the design and implementation of our QoS-aware DSMS that we dub as SpatialDSMS (short for Spatial Data Stream Management System), which operates over fast arriving geo-referenced data streams. Our system receives input data from either a stream source or a batch source. Through an appropriate interface, we provide the user with the ability to express their queries (i.e., batch or continuous) and serve them together with QoS budgets to the system. Budgets are expressed as QoS goals (e.g., latency/throughput, estimation quality or resource utilization). Thereafter, our system ensures that the query is

executed within the specified budget and results are served to the user either incrementally (in case of online processing mode) with rigorous error-bounds or as only-once (in case of batch processing mode). Streaming data sources can be combined, on-need, with a batch static data (in what is known as stream-static join) to answer an interactive query that requires enrichment with master data (i.e., disk-resident data).

In this thesis, we show the following constituting parts that collectively form our system (i.e., SpatialDSMS).

1.3.1 SpatialBPE and SpatialNoSQL: Scalable Distributed Spatial Batch Query Processing and Storage

We begin this thesis by designing two QoS-aware custom data partitioning methods and their associated query optimizers for scalable storage and batch processing of big spatial (we use spatial and geospatial interchangeably hereafter) data. We dub those systems as SpatialNoSQL and SpatialBPE, respectively. Spatial data partitioning is a mean-to-an-end, where the goal is achieving quality goals; lowering latency and maximizing resource utilization while keeping accuracy levels high. To achieve those, we design Geospatial Sharding Scheme (GSS), a custom spatial partitioning method for a NoSQL scalable distributed storage emerging framework, MongoDB [2] , together with a query optimizer that exploits GSS for improving the quality of service, both constituting SpatialNoSQL. We also have designed a custom spatially-attuned adaptive partitioning method that we dub as SCAP, which adequately trade-off three contradicting spatial partitioning goals (i.e., boundary spatial objects - a.k.a. edge cases, spatial co-locality preservation and load balancing) in an emerging batch processing framework (i.e., Apache Spark [1]). We further have retrofitted a density-based clustering algorithm so that it exploits SCAP, both the SCAP and the associated query optimizer form SpatialBPE. We have evaluated SpatialNoSQL and SpatialBPE using real-world geospatial big data loads. Our results show that SpatialBPE and SpatialNoSQL outperform state-of-art counterparts by significant magnitudes. Also, they were able to meet QoS goals specified as latency/throughput and resource utilization. SpatialNoSQL is geared toward scalable distributed storage, whereas SpatialBPE is designed for distributed batch processing.

1.3.2 SpatialSPE: Spatial Approximate Query Processing

After designing spatial-aware appropriate data partitioning and query optimizers for distributed scalable storage and batch processing, we have realized that those methods alone cannot achieve QoS goals for spatial interactive analytics, where fast arriving fluctuating (i.e., in skewness and arrival rate) spatial data streams hit so hard the resources of the SpatialDSMS. Also, we aim at a system that supports incrementalization of spatial data stream computation, meaning that results are served incrementally based on time-based window semantics without the need to recompute or materialize previous loads. to achieve those goals, we design SpatialSPE, which aims at achieving low-latency and maximal resource utilization, while serving results with acceptable high accuracy expressed as rigorous error-bounds. SpatialSPE accepts a continuous query and QoS budgets (expressed as latency and accuracy targets) and employs our spatial-aware sampling method (that we dub as SAOS, which is an integral part of SpatialSPE) to select an appropriate sample, then it computes an approximate answer and serves it to the user incrementally, together with rigorous error bounds. We have implemented SpatialSPE on top of Spark Structured Streaming [6]. Our results show that SpatialSPE (and the incorporated SAOS scheme) outperforms baselines by significant margins. Also, combining SpatialSPE with the partitioning methods (GSS from SpatialBPE and SCAP from SpatialNoSQL) is possible in order to materialize (partial) stream data loads efficiently for a future (semi-)batch processing, rendering them complementary, and the combination allows to benefit from both worlds without their limitations.

1.3.3 SpatialSSJP: Adaptive Stream-Static Spatial Join Processing

After designing SpatialBPE, SpatialNoSQL and SpatialSPE, we have realized that interesting mixed-workloads in highly dynamic environments require combining batch and streaming views (i.e., current views with historical views) or enriching spatial streams with static descriptions. For this purpose, we have designed SpatialSSJP, a QoS-aware adaptive stream-static join processor that exploits SpatialSPE (and specifically SAOS) in adaptively selecting proportionate sampling fraction through the application of an embedded rate controller and serve it to SAOS using a feedback loop mechanism. SpatialSSJP is an approximation framework that is designed to efficiently tradeoff miniscule error-bounded accuracy for low-latency, thereby assisting SpatialDSMS to survive during brutal burst

spikes in data arrival rates. SpatialSSJP achieves that in a circadian rhythm without compromising the overall stability of SpatialDSMS. We have implemented SpatialSSJP on top of Spark Structured Streaming [6] to complement SpatialSPE for approximate query processing of fast arriving spatial big data loads. Our evaluations with real-world scenarios and big spatial benchmarks and data loads prove that SpatialSSJP is able to survive even the most striking burst workloads while keeping accuracy loss in check (i.e., under a statistically desirable margin).

1.4 Thesis Outline

This thesis is organized in the following chapters.

In **Chapter 2**, we show a background about data processing in dynamic and scalable applications and big data management frameworks that have been exploited in this thesis.

In **Chapter 3**, we show the overall architectural design of our system SpatialDSMS.

In **Chapter 4**, we show the design and realization of SpatialBPE and SpatialNoSQL, two quality of service aware frameworks for distributed batch processing and scalable storage, respectively.

In **Chapter 5**, we show the design and realization of SpatialSPE, a Spatial Approximate Query Processing engine.

In **Chapter 6**, we show the design and realization of SpatialSSJP, an adaptive Stream-Static Spatial Join Processing system.

To sum up, in **Chapter 7**, we conclude our works, showing the implication that can be carried over to other domains, and some future research frontiers.

Chapter 2

Background

In this chapter, we start in § 2.1 by showing a baseline architecture for DSMSs that has gained a momentum in the last decade. We then showcase the capabilities of big data storage (§ 2.2) and analytics (§ 2.3) frameworks that we have exploited to implement our algorithms and systems that we are presenting in this thesis.

2.1 Lambda Architecture

Challenges associated with managing mixed streaming big data workloads have motivated the emergence of novel dynamic architectural patterns such as the Lambda architecture [7]. The Lambda architecture employs real-time stream processing for timely approximate results and batch processing for delayed accurate results. Figure 2.1 shows a typical Lambda architecture.

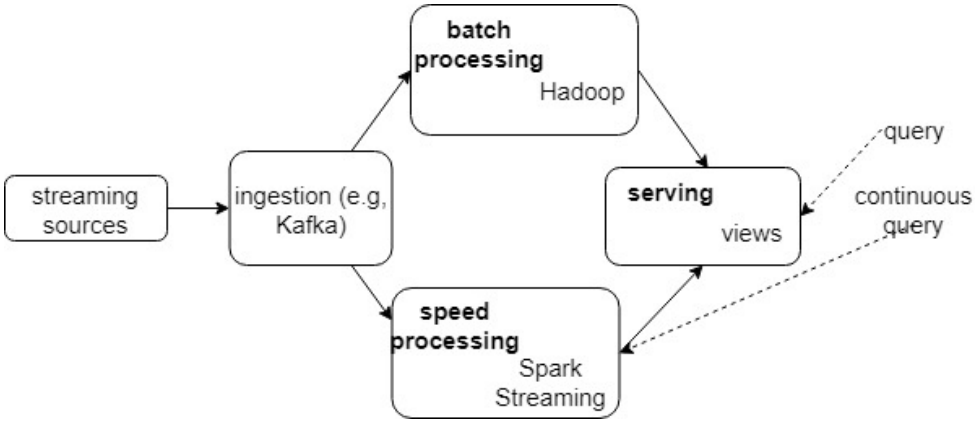

Figure 2.1. Typical Lambda architecture

New streaming data is served to either a batch layer or a speed layer. Accurate, often computationally expensive, posterior analytics are performed on historical data (a.k.a. data-at-rest) in the batch layer (e.g., using Spark). On the contrary, approximate queries are performed in the speed layer, analyzing and processing stream data (a.k.a. non-stationary) on-the-fly (e.g., using Spark Streaming). Mixing workloads in this setting means basically exploiting static historical archives from the batch layer in predicting future or current trends

in the speed layer, and thereby accelerating the processing and possibly helping in the prediction of a sudden brutal burst spike in an arriving data load (i.e., being proactive). Stated another way, batch layer serves as a synergistically complementary processing engine that performs complex computations (which are prohibitively expensive online, such as a deep learning model) on static data (i.e., collected previously from active streams) aiming to gain deeper insights , correlations and patterns, which together with the help of online analytics serve a clearer picture that better facilitates timely decision making. In this sense also, serving instantaneously two paths of computation better helps in resolving the cause/effect problems, where a speed layer can appropriately discover an effect that is explained by a deeper (i.e., resource-intensive and costly) analysis in the batch layer. Views are normally served through the serving layer (e.g., using MongoDB). Representative frameworks that can collectively form a typical ecosystem based on the Lambda architecture are discussed in the next subsections.

2.2 Distributed (Spatial) Big Data Storage Frameworks

In this section, we summarize the features and traits of an emerging scalable distributed storage NoSQL system that we have exploited for building up SpatialNoSQL (one of our sub-systems) as discussed in section 4.8.

2.2.1 MongoDB: A Scalable Distributed Storage Framework

MongoDB is a document-oriented scalable NoSQL distributed (thus simplifying horizontal scaling) database management system that offers many indexing strategies for a highly-performing batch processing experience. Data is stored in a flexible changeable JSON-alike representation that offers freedom in dynamically changing the data structure to be able to gather heterogeneous data sources under one umbrella. In MongoDB terms, each document contains key/value pairs of an entity, and several documents (analogous to records in RDBMSs) constitute a collection (analogous to tables in RDBMSs).

In RDBMSs, information related to an entity are normally spread out between many tables and are collected through their referential integrities (i.e., the relations represented through primary/foreign keys) at run time. On the contrary, in MongoDB, the notion of referential integrity vanishes. To compensate for that, MongoDB is using an *embedded document*

metaphor, where documents are values for keys, thus naturally organizing data better than the plain flat structure (i.e., key/value pairs) [8].

Architecturally speaking, MongoDB is built to operate on sharded clusters (analogous to master/slave architectures), where one (or more) routers (*mongos* in MongoDB terms) shard (split) data points to several (two or more) parallelly connected shards (analogous to worker nodes, slaves or executors) aiming at distributing the load so as to provide a scalable storage for massive amounts of datasets, and thereby demystifying the access for analytics.

2.2.2 Geospatial Analytics in MongoDB

MongoDB supports primitive types of spatial operations and associated access structures (i.e., indexes). It natively supports two types of geospatial indexing; 2dsphere and 2d. 2dsphere is designed for spherical geometries, whereas 2d indexing flattens the earth out (similar to a heuristic overview of a grid, two-dimensional Euclidean plane) [9]. 2dsphere yields more accurate results than 2d because of the representation, where the latter is used for queries that use flat geometry as it assumes a perfectly flat surface, thus causing (massive) distortions near the earth poles. 2d supports the "$geoWithin" and "$near" operators.

Several geospatial queries are supported, including proximity (i.e., nearness, through $geoNear operator, an aggregation operator), intersection, or inclusion (i.e., 'within' predicate) by providing appropriate operators such as "$geoWithin". Those queries are supported for geospatial points and shapes (i.e., line, polygon).

$geoWithin is normally utilized to search for geospatial points within a shape (represented on a flat surface, such as a rectangle, polygon, or a circle)

We have selected MongoDB in this thesis as a baseline representative to base some of our batch-oriented implementations because of the spatially-oriented overarching support it offers natively. We have stacked-up SpatialNoSQL specifically over MongoDB (as explained in chapter 4)

2.3 Distributed (Spatial) Big Data Processing Frameworks

Another important component that resides in the batch layer of the lambda architecture is distributed processing frameworks such as Apache Hadoop [10] and Spark (patterned after

Hadoop MapReduce module). Of late, Spark has shown superiority over Hadoop and most efforts of the relevant literature are stacking up on Spark. Spark significantly outperforms Hadoop especially for iterative structures that access in-memory data excessively. This encouraged us to focus on Spark exclusively as a big data processing framework in this thesis.

We first list basic features of Spark core, with some interesting traits that encouraged us to favor it over counterparts as a representative for stacking up our algorithms. We then shortly recapitulate spatial-aware plugins that have been patterned on Spark such as GeoSpark [11]. Thereafter, we overview (spatial) stream processing frameworks, specifically Spark Magellan[1] [12, 13]. We stack up SpatialBPE (chapter 4) SpatialSPE (chapter 5) and SpatialSSJP (chapter 6) on Spark's Magellan plugins to realize our standard compliant prototypes.

2.3.1 Batch Processing: Apache Spark

Apache Spark [1] is an open-source framework that has been patterned after MapReduce framework, aiming at processing huge amounts of data efficiently in parallel computing environments. It is an efficient general-purpose solution for processing disk-resident, memory-resident and big data streams (in micro-batching mode). The core programming abstractions of Spark are RDDs [14] , which are groups of objects partitioned across multiple computing resources for parallel manipulation, where each partition containing an RDD is processed parallelly in a single task. Spark jobs include constructing new RDDs, RDD's transformations (i.e., filter and map), which are performed on a coarse-grained fashion, or calculating a result by invoking a function on RDDs (a.k.a. action in Spark's jargon, such as *count* or other *reduce* functions).

Spark provides high-level APIs for various programming languages such as Java, Scala, and Python. It allows programmers to develop a complex data pipeline system, parallelizing multiple processing flows through the Directed Acyclic Graph (DAG) pattern.

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

Spark executes the lineage DAG graph lazily in such a way that transformations are performed only after encountering an action in the graph. RDD is constructed either from scratch or through a transformation from one RDD into another. Spark architecture is master/slave where the master controller receives the result of a DAG after completion [15].

The technical burdens brought by RDD-based operations hinder a wider adoption of the Spark between non-technologically-experienced users. This has motivated the emergence of a full-fledged SQL-alike API that demystifies such an adoption, initiating by its batch format , Spark SQL [16] , that served as a precursor to launching a streaming version, Spark Structured Streaming (SpSS for the most of the remaining of the discussion hereafter). This declarative SQL-alike support introduces *DataFrames* and *Datasets* to represent distributed collections, with additional schema information (as opposed to RDDs) [15].

Spark default join on RDDs is *shuffled hash join*, implemented through *cogroup* operation [15, 17] (analogous to the hash-partitioned join, and similar to full outer join in SQL), which requires shuffling of both input RDDs in case that partitioner is unknown for both. Spark needs the data that has same keys to reside on the same partition to be joined. However, in cases where one RDD has an associated partitioner, it needs no shuffling, instead the other RDD is shuffled with the same partitioner so that its elements hit the same node hosting the partition of the other RDD and collocate for the join operation to proceed.

There are two types of join in distributed environments; *broadcast* and *repartitioning* join. The former is possible in cases where one of the RDDs (and similarly the DataFrames in Spark SQL) fits in main memory of the worker nodes. In such a case, it is broadcasted to those nodes, what then remains incumbent is a map-side combine with each partition of the larger RDD [15] (and similarly the DataFrames in Spark SQL). In the latter case, where RDDs (and similarly DataFrames) do not fit in the fast memory, a repartitioning join is performed, which requires shuffling as explained earlier. In Spark SQL, broadcast join is configurable through enabling/disabling "*autoBroadcastJoinThreshold*" (enabled by default). It worth mentioning that joining with non-unique keys result in a costly cross product.

Limiting the comparison to a single plain ecosystem felt all wrong. So now we're flipping the switch on some non-trivial architectural tiered plugins that make distributed in-memory

spatial analytics a reality. Even though Spark outperforms its predecessors for processing big data, it is still not optimized for specific application scenarios, such as geospatial data analysis, which led to the emergence of spatial-aware extensions built on top of Spark core. Two recent representatives are GeoSpark [11] and Spark's Magellan [12, 13]. Both have seen swift adoption throughout the Spark community.

I) **GeoSpark** [11] is a spatial-aware open-source framework that have been designed specifically for processing massive amounts of spatial data loads. it has been engineered atop the Spark's pyramid, extending the traditional Spark core layers with spatial-aware abstractions and counterparts. For example, GeoSpark has extended the Spark abstraction RDD into a spatial-contemporary that is termed as SpatialRDD (SRDD for short), signifying that it preserves the idea of the RDD abstraction but introducing the multidimensionality to the equation. GeoSpark supports a myriad of spatial operations and predicates, most importantly, *k*NN, ranges searches and spatial join. The architecture of GeoSpark is explained in Appendix A.

I) **Spark Magellan**[2] [12, 13] . Magellan is the first-in-class library that is fully extending Spark SQL declarative API by offering a layer of geospatial analytics relational abstractions. It offers a developer-friendly interface that allows executing spatial query primitive in a QoS-aware fashion, focusing mainly on low-latency and high throughput. Magellan optimizes the query plan by offering low-cost spatial indexing.

At a cursory level, Magellan is layered on Spark, which by itself is the de-facto standard for big data processing so far and looks set to remain that way at least for the foreseeable future. Using the z-curves (in addition to non-hierarchical grids), the filtering step reduces to that of point-in-MBR (a.k.a. MBR-join) test, which is computationally plausible. Magellan join algorithm obeys the true hit filtering approach [18, 19] (specifically filter-and-refine approach). Magellan supports several spatial predicates including intersection, containment

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

and inclusion (i.e., within). Spatial join is supported natively and both participating relations are indexed with z-order curves.

Magellan natively operates on a costly cross join. However, it has incorporated an optimization for performing an inner join instead. It achieves this by indexing points and polygons using geohashing (a special case of z-order curves based on bounding boxes), then performing a hash join (i.e., filter stage) that is proceeded by a PIP test (e.g. a '*within'* predicate, refinement stage), thus discarding possible false positives (i.e., cases when geohash bounding box boundaries of a query point intersect a polygon, but query point falls actually outside). Listing 2.1 shows an example PIP query using Magellan.

```
pointsDF.join(polygonsDF,pointsDF("index")==
polygonsDF("index")).where($"point" within $"polygon")
```

listing 2.1. Example PIP test in Magellan

In this case, a quick-and-dirty sieve (filter) is first applied (pointsDF("index") == polygonsDF("index")) that is really a cheap hash join on the index, resembling the filter stage of the filter-and-refine approach, thereafter costly PIP test ($"point" within $"polygon") is applied to discard false positives.

Spark's Magellan automatically stores z-order curves that cover a geometry (i.e. polygons representing neighborhoods or counties in a city), with the associated relation (e.g., '*contains'*, '*within'* (contained in), '*intersects*') expressing the relation between the z-order curve (geohash for longitude/latitude representations) and the geometry. This relation is important to minimize the costly '*within'* predicate (i.e., PIP test), which will be evaluated only when a z-order curve that is enveloping a query point is not guaranteed to fall within a polygon.

Most importantly, Spark's Magellan supports PIP test (i.e., geofencing). Being stacked up over Spark SQL, broadcasting is by default enabled, meaning that a small DataFrame (representing polygons in the PIP test input) will, by default, be broadcasted to executor nodes. Join key in this case is a geometric field (i.e., multidimensional), and Spark natively does not support partitioning based on such keys. Hash partitioner is the default used by DataFrames, which takes the hash value of the join key and calculates the modulus of dividing it by the number of partitions, then it emits the tuple to the partition that is corresponding to the resulting value. This means that if we are able to reduce the multidimensional representation of a geospatial object into one-dimensional space, then geometrically-nearby objects should end up in the same partition.

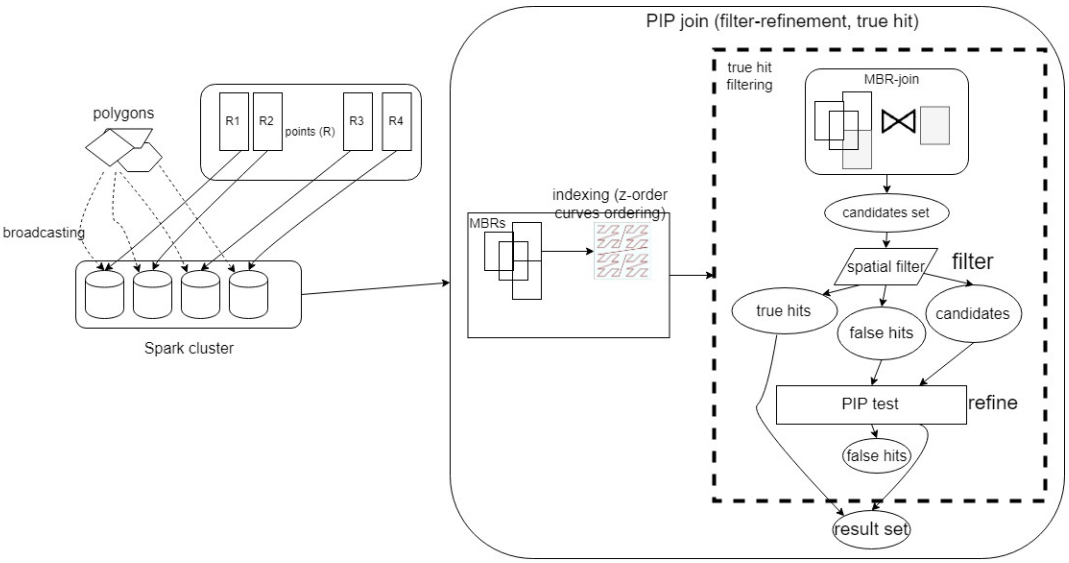

Figure 2.2. PIP test in Spark's Magellan, Filter-and-refine (true-hit part) is adapted from [20]

Figure 2.2 represents a high-level sketch of a general structure of PIP join performed in Magellan, which also serves as a machine for elucidating the broadcast join mechanism in distributed systems, in addition to the true hit filtering join approach [20].

Magellan and GeoSpark are not panacea but instead are springboards to begin with QoS-aware spatial optimizations.

We found Magellan superior to GeoSpark and other spatial-supporting frameworks in the sense that Magellan is built with the spark fluent API (which allows wiring up all functions in a single expression) in mind. All spatial operations that has been pushed up the stack are

obeying this fluency, thus complementing the Spark's modular hot-swappable architecture, and thereby avoiding to reason about the underlying processes atomically (as recommended by Spark's development team [6]), which is one of the main design goals targeted in Spark. Also, Magellan offers access structures (i.e., indexing schemes) that have less associated computational complexity, such as z-curves.

2.3.2 Online Processing: Spark (Structured) Streaming

Stream Processing Engines (SPE) are machines designed to process avalanches of fast arriving unbounded online data streams, aiming at gaining deep insightful views that support decision making and strategic planning in real time. They normally employ a graph of operators (typically a DAG) where operator instances are distributed to parallelly connected processing nodes so as to accelerate the processing, which is normally incremental, meaning that results are dynamically updated as new data arrives. A major challenge in distributed stream processing is the state management, where intermediate computation states need to be stored/retrieved in a consistent manner that does not deteriorate the benefits of parallelization.

The distinction between batch and online processing modes is that in the latter a push mechanism is applied where data is pushed by sources to be processed by an SPE, whereas in the former a pulling mechanism is applied such that a system pulls data residing in disk. Also, batch processing is typically an exactly-once operation, whereas online mode runs endlessly and compute results stepwise.

Second, due to the fact that within a DSMS, data must be processed in a push-based manner, the temporal aspect of the query execution is more important than that of pull-based query executions in a database system.

Those distinctions impose challenges that normally do not affect batch processing systems. Queries run against a data stream are known as Continuous Queries (CQ) (sometimes colloquially referred to as online querying, termed continuous as opposed to one-time queries) that run in unbounded fashion, hence the order of data arrival is focal and is normally accounted for by windows semantics (i.e., temporal intervals bounding the start/end times of every query execution trigger) [21]. Also, as clock ticks forward, stream data becomes obsolete and potentially loses its value, hence low-latency is a priority QoS goal. In addition,

stream data normally exhibits temporal skewness and fluctuation in the arrival rates that alternate between brutal spikes and underloads, and a good SPE should be able to survive such unpredictable behaviors. SPEs store arriving tuples and access structures (i.e., indexes) in-memory for speeding up the processing.

Spark Streaming [22] is a SPE that splits arriving stream tuples into blocks of RDDs dubbed as discretized streams (or D-Streams) based on the time-based window semantics, where every batch interval, micro-batches that are comprised of RDDs are sent to the batch Spark processor in a process that is termed as micro-batching. Most transformations supported on RDDs are also supported on D-Streams [15].

Spark Streaming does not natively support the join between streaming data and static relation. It is otherwise supported in Spark Structured Streaming (explained shortly). Spark Streaming is robust against arrival rate fluctuations for aggregation queries [23] . Also, Spark Streaming, which uses the micro-batch model, is more fault-tolerant than Storm and Flink (which are otherwise brittle and easily prone to failures), which use the record-at-a-time model [24], which acts on per-row basis. Upon nodes failure, micro-batch based systems recompute lost data efficiently [25] , thus recovering quickly [24, 25] , which is a plausible overarching trait that is expensive in systems that obey record-at-a-time model, simply since those models require serial replay processing [25] . Also, binding schemas to data sources is straightforward in Spark SQL, which demystifies transforming it into a Scala case class, thus enabling type-safe querying, which is a plausible trait while jumping through operating machines. All those traits provided by the micro-batching model encouraged us to adopt it for our implementations.

Spark Structured Streaming [6] (hereafter SpSS for short) is a new layer atop Apache Spark layered-up ecosystem. It is a high-level API that lends many concepts from the original Spark Streaming design [25] . Structured streaming mainly differs from the discretized streams in that users normally express queries using a declarative SQL-alike API (in the form of DataFrames [16]) instead of manually building a pipeline DAG of operators (such as those found in MapReduce). Being a newly addition joining Spark’s family, it lacks accompanied proper documentation for many details explaining specifically its internals and subtleties. As

such, we here provide a structure that captures the anatomy of major constituent parts and their workflow.

SpSS aims basically at resolving latency and accuracy issues that are normally confronted in end-to-end data analytical deployments, where, more than often, fast arriving data torrents read on-the-wire are joined with batch tables for interactive insightful BI analytics. Production pipelines depend on joining serving streaming system's workloads with transactional ones, while most SPEs currently focus on streaming computations, spark structured streaming is gaining more attention because it places due importance on other batch loads combined with streaming, thus improving the end-to-end performance in time-sensitive production systems. Another outstanding feature that makes structured streaming favorable over counterparts is the *incrementalization* in the continuous query execution model. This means at a highest level that users write their queries as if they were to be executed in a batch mode and SpSS incremantalizes those queries to be executed in a streaming mode with no further effort from the user's side, thus relieving the overburdened load from the shoulders of users and programmers from reasoning about the underlying mechanism of such an optimization. SpSS was able to achieve that by reusing the Spark SQL optimizers [16] (such as Catalyst, which utilizes advanced features such as Scala pattern matching in a distinguishable manner that improves the query optimizer and makes it extensible), which streamlines the adoption of any newly added SQL batch functionality in the future. The default mode of operation is micro-batching via fine-grained tasks [1] (i.e., using the discretized streaming execution model from Spark Streaming). 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 infinite input table. User expresses a batch-alike query and the underlying SpSS engine translates that into an incremental query scheduled to be executed on the infinite input table. Results in the result table are updated based on a trigger (analogous to batch interval in Spark Streaming).

Likewise, SpSS uses fluent API that enables users to chain pipeline operators, allowing them to describe the input data source using *method chaining*, aiming at an enhanced code readability in a way that resembles a sequential written prose. We try here to uncover the peculiarity by which D-Streams and SpSS are operating internally throughout a unique anatomy which is elucidated in figure 2.3.

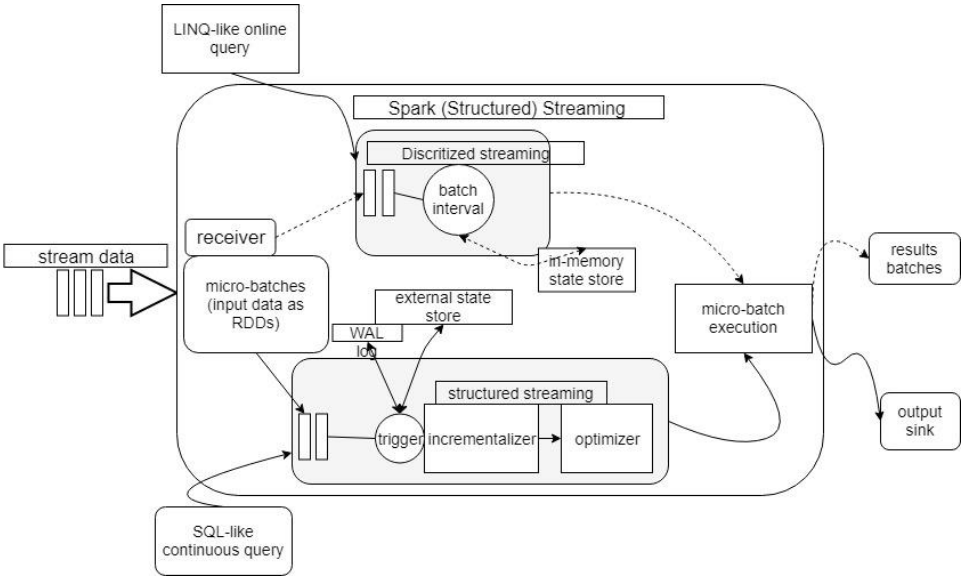

Figure 2.3. Anatomy of Spark (Structured) Streaming

Also, one more distinguished feature of SpSS is that it can easily manage stateful aggregations, not to be confused with batch aggregations, as the former means those aggregations with states that are incrementally evolving over time, in interactive settings (such as 'counting by groups'), whereas the latter means having a single value (such as '*count*', '*sum'*) computed in a static batch mode . As we can see from figure 2.3, SpSS handles stateful aggregations by keeping aggregation states and midway results in fault-tolerant state store (every worker node has its own state store instance), so as to invoke the state at every trigger and build aggregations upon it incrementally. By doing so, SpSS can continue from where it left off upon any non-intentional system crash. Internally, at every trigger, the computation of stateful aggregations compiles down to that of a MapReduce Spark job.

All those traits make Spark (with all its constituting parts) a perfect match for complementing our architecture, achieving basically an important design goal of being able to apply the same

programming efforts in batch and interactive modes. In this thesis, we have decided to opt for SpSS as a reference system, since it outperforms counterparts (such as Apache Flink and Kafka Streams) by orders of magnitudes [6] .

To our knowledge, there is no consolidated framework or system for managing streaming geospatial datasets. Existing systems, such as GeoSpark and Spark's Magellan are designed to operate in batch modes (better suiting the batch and serving layers of Lambda-obeying architectures). All other efforts are either ad-hoc fixes or patches and glues that do not collectively form a comprehensive framework. However, libraries and frameworks such as Magellan and GeoSpark are compatible jumping off points for initiating a constellation of optimizations and a new breed of contributions toward a full-fledge online spatial processing engine. More theoretically, since, for example, Magellan is built on Spark SQL and since the optimizer of SpSS accepts a batch-alike query and automatically incrementalizes it on the unbounded input table, it is then evident that Magellan is a candidate for an optimization and can be efficiently retrofitted for spatial interactive queries (such as spatial online join processing). This is a significant contribution of this thesis as will be explained in chapter 6. In the next chapter, we explain in detail the general architecture of our system.

Chapter 3

SpatialDSMS: Spatial Data Stream Management System

In this chapter, we start in § 3.1 by showing the type of analytics that novel systems should provide to be able to cope up with the QoS-demanding requirements of highly dynamic and scalable applications. We then, in § 3.2, explain the QoS attributes that we are supporting in our system, including a general methodology for measuring the accomplishment of QoS goals through services provided by our system. Thereafter, in § 3.3, we recapitulate the importance of fusing scalable storage with fast analytics, promoting our architecture which we then introduce in § 3.4.

3.1 Spatial Data Analytics in Highly Dynamic and Scalable Applications

Applications in smart cities, Industrial Internet of Things (IIoT) and Industry 4.0 demand an awareness of specific dimensions that have been long treated as second-class citizens. Most dynamic applications nowadays are focusing specifically on location, where extra locational information offers support for optimized deep insightful exploration of data that leads to improving the overall quality of the service an information system is offering, aiming ultimately at enhancing the quality of our lives in many aspects. The abundance of geospatial data streams has motivated several new application scenarios that would remain otherwise illusive. For example, a system for road traffic control that has been proposed by [26], which aims at lowering congestions and improving future city planning in a way that lowers the toxic emissions from vehicles. Other examples include, designing reactive and proactive solutions for monitoring environmental crises, such as hurricanes, animal herds and oil-spills [27] , air quality and pollen distribution [28]. Also, providing personalized location-based services (LBS) through the exploitation of location metadata in social networks [29]. Additionally, fusing social data, such as tweets from the micro-blog service *Twitter*, together with trajectory data collected through GPS-enabled devices, in a data mining algorithm to cluster topics discussed by region [30] or visualize (by exploiting heat maps) planetary-scale or city-wide scale distributions of people communication activities [31] . In the same vein, [32] have designed applications for finding local Twitter influencers to detect local events by tracking their tweets. Moreover, complex mixed workload application scenarios, such as

applying representation learning for the analysis of cars driving behaviors [33], improving the bike sharing experience [33], and location-based recommendations [29].

What is then axiomatic in all those dynamic applications is that they require the acquisition of diverse spatial analytics. All query types related to location intelligence (a.k.a. spatial intelligence) are receiving more attention in the last decade or so. Our application scenario, discussed in section 1.1, requires passing through an end-to-end QoS aware spatial data processing system.

Location intelligence is the process of deriving meaningful insights from geospatial data relationships, modelling the interaction of spatial objects with their surrounding ambient [34]. In achieving this goal, many spatial queries are common, from the simplest forms all the way up the pyramid to the most complex composable queries. Integrating Business Intelligence (BI) with location data has long history in yielding better ad hoc reporting experiences that benefits those businesses. The essence of this intelligence is composed of the capacity to organize complex huge data in a way that exploits geographical information in revealing hidden relationships between locations and events. Moreover, dynamic applications in smart cities are depending on visualizations (for example, heat maps) to understand hidden patterns in the data that are not normally shown through traditional tabular formats. Visualizations and other forms of dashboarding are the ultimate goals. However, reaching that point requires a spatially attuned end-to-end data stream management system that constitutes in-between transformations and analytics, which then resembles the architecture of our system SpatialDSMS (introduced shortly in § 3.4). In this section, we identify the most recurrent types of spatial analytics that can be executed in either one of two modes, batch or online, which collectively provide baselines that can be efficiently exploited in en-route to achieve locational intelligence with quality guarantees.

The following is a list of the most common geospatial queries that we natively support in SpatialDSMS:

1) **Range spatial query (**a.k.a. **proximity queries)**. Range searches return the set of spatial objects that fall at a maximum specified range (e.g., radius) from a specific spatial object (most often referred to as focal point, query point or test point). An example spatial range search from our scenario is “finding people near

an accident location in range that is equal to 1K meters maximum". We support range spatial queries for the batch processing (explained in chapter 4).

2) **Spatial join**. In its general form, spatial join is a set of all pairs that is formed by pairing two geo-referenced datasets while applying a spatial predicate (e.g., intersection, inclusion, etc.,) [35]. The two participating sets can be representing multidimensional spatial objects. An example spatial join query from our scenario in section 1.1 is "finding boroughs to which each GPS-represented spatial point (volunteer) belongs, a.k.a. geofencing", which requires joining spatial points with a master table representing boroughs.

   In mathematic terms, given two sets A and B, a spatial join returns a set of pairs (a, b) that satisfy the formulation in (3.1)

   $A \bowtie_{pred} B = \{(a, b) \mid a \in A, b \in B, pred(a, b) == true\}$. (3.1)

   , where $pred$ is the spatial predicate applied (e.g., touches, intersects, overlaps, etc.,). It worth mentioning that since proximity ordering is not preserved with digitized representations of spatial objects that are candidates of a join, relational join methods such as sort-merge join are not applicable. Also, equijoin (e.g., hash joins) is generally inapplicable in cases where spatial objects that are involved have extents. This can be mitigated with dimensionality reduction approaches that impose a spatial ordering, such as the application of z-order curves, thus projecting spatial objects into one-dimensional space (more about this in chapter 6).

   Checking the join condition (a.k.a. predicate, such as 'intersects', 'touches', 'within', 'contain, 'overlap') is an expensive operation. As such, most well-performing algorithms employ a two-stages approach that constitutes *filtering* and *refinement* (patterned after true-hit filtering approach [20]). The former aims at pruning the search space by first applying a quick-and-dirty sieve (filter), performing a spatial join on approximations of the objects (typically MBRs, known as MBR-join [20]). In the refinement stage, incorrect results (i.e., false positives) caused by the approximations are removed using the exact geometry processor (i.e., the expensive predicate) that is applied on the [20]original objects. Spatial refinement dominates the cost of the whole join procedure, thus designs

should consider minimizing edge cases (we refer to those as Boundary Spatial Objects (BSO) in chapter 4) so as to relax the cost induced by applying it. Spatial join is a primitive that acts as a pulsating heart in dynamic application scenarios that normally require intermixing geo-referenced datasets for deeper analytics. More on spatial join in chapter 6.

A special case of spatial join is represented through containment (or inclusion) test that seeks whether a spatial object falls within the boundaries of the extent of another object or outside.

We support two types of spatial join; static-static (i.e., deterministic), within the layers of SpatialNoSQL as explained in chapter 4, and stream-static (i.e., probabilistic), within the layers of SpatialSSJP as explained in chapter 6.

3) **Spatial clustering.** Clustering algorithms basically aim at grouping identical spatial objects together into subgroups called *clusters*. From many types of clustering algorithms, density-based clustering [36] has picked up pace recently and is widely accepted for the overarching traits it provides. It is a class of clustering that basically works by separating spatially dense space regions from outliers, thus dense regions constitute clusters. A well-known method for density-based clustering is DBSCAN [37] . However, tailoring such an algorithm for the parallel computing environments requires attention, as a naïve solution poses heavy network communication overhead. To cope with this challenge, related versions (DBSCAN-MR [38] or MR-DBSCAN [39]) have been tuned for parallel general-purpose big data workloads. Clustering is one of the most important data analytics activities [40] . We support density-based clustering within the layers of SpatialBPE as explained in chapter 4. An example spatial clustering query form our scenario in section 1.1 is “grouping volunteers, in specific proximity to incident location, by the level of training they possess”
4) **Spatial geo-statistics.** We support two types of spatial geo-statistics. Those are Linear (a.k.a. single queries) and online aggregations (e.g., top-N). computing those queries in batch mode is straightforward. We alternatively aim at optimizing their execution in the streaming (i.e., online) mode, where we incrementalize the results of computing those queries, considering an unbounded

input stream. Incrementalizing those queries requires special attention specially for ensembles (e.g., Top-N) that normally encapsulate an online aggregation, which requires a costly state management. For single queries, we support statistics such as 'average' and 'total' of target variables as primitives. Other statistics can be estimated based on those primitives. An example spatial statistic query form our scenario in section 1.1 is "finding the average trip distance travelled by ambulances originating from specific regions in the city and ordering them in descending way". We support spatial statistics within the layers of SpatialSPE (the topic of chapter 5) and SpatialSSJP (the topic of chapter 6).

5) **K-nearest neighborhoods (kNN).** It is an optimization proximity search problem (i.e., based on range search queries). Formally, given a set A of points in an embedding space S and a query point (a.k.a. test point) $q \in S$, *k*NN seeks to find the $c \geq 1$ number of points forming a subset B such that all points in B are closest than all other points in the remaining subset ($A - B$). Stated another way, every point in A but not in B is at least as far away from q as the furthest point in B. More mathematically, given a query point q, a set of $c \geq 1$ nearest neighbor to q is B, where $B \subseteq A$ such that $||S|| = c$ and $\forall$ point $p_i \in (A - B)$, EuclideanDistance $(q, p_i) \geq \max_{qp \in B}(q, qp)$. We support *k*NN for batch mode within the layers of SpatialNoSQL as explained in chapter 4. An example *k*NN query form our scenario in section 1.1 is "finding the nearest 10 volunteers around an incident location".

Other primitives that we do not support natively but are easily composable from our baseline primitives include the following:

6) ***k*NN join.** *k*NN join sets on the confluence between *k*NN and spatial join. Formally, having two geo-referenced datasets A and B, *k*NN join generates $c \geq 1$ closest neighbors in B for every object in A. More theoretically expressed in (3.2).

$$A \ltimes_{kNN} B = \{(a, b) \mid \forall a \in A, \forall b \in B, kNN\ (a, b, k) \text{ is true}\}. \quad (3.2)$$

In other terms, *k*NN join can be loosely defined as finding all *k*NN objects (belonging to a spatial data set) for every object of another spatial set. This operation is extremely expensive as it combines the complexities of two complex

spatial query processing operations, spatial join and *k*NN. An example *k*NN-join from our scenario (recap section 1.1) is "selecting k-nearest well-trained passing-by medical staff members and ordering them by their location in relative to many incidents having emergencies at same time". This intrinsically encapsulates two spatial datasets, volunteers and locations of many incidents (or patients with sudden health problem attack), such that for every incident the algorithm selects k-nearest volunteers that satisfy all spatial query predicates.

*k*NN-join traditionally constitute three main steps. Those are data partition, candidate selection and *k*NN join steps, which can be realized with MapReduce [41]. Authors apply Voronoi diagrams as a tessellation method in the partitioning step (mostly a map transformation), whereas the candidate selection step constitutes some algebraic calculations based on the Euclidean distance basically, then a join is applied (a reduce action in Spark terms) to join the candidate set with partitions. We do not natively support *k*NN join, but we provide all the QoS-aware spatial analytical primitives for easily constructing an efficient *k*NN join algorithm.

We directly support baseline spatial analytics primitives that are the most important of the myriad of spatial data analysis activities. We also posit that other workloads are composable and can be efficiently stacked-up the pyramid. By those supports, we aim at a modular system design to manage streaming spatial data in a coherent way. To achieve this goal, we have designed SpatialDSMS, comprising highly-efficient algorithms in batch static modes and in streaming modes where data arrives in a high pace into the system. In the next subsection, we define the most recurrent QoS attributes that we support in our system.

3.2 Quality of Service Goals

In life-critical applications such as healthcare, it is very important that services provided by a data management engine meet a prespecified set of SLAs that intrinsically encapsulate QoS goals. Common metrics of the performance of a DSMSs in meeting QoS requirements include, most importantly, latency/throughput, accuracy and resource utilization. Quality attributes constraint system functionalities, specifying a qualification (a.k.a. annotation) on

how those functions are performed. Such as constraining a spatial query to be performed with a low-latency.

Distributed big data management systems should treat QoS-awareness as a first-class citizen when designing their services, such that they serve in accordance with QoS properties of the SLAs. Achieving this goal is specifically challenging as it necessitates intelligently trading off several contradicting factors. A problem that is further inflated when operating in a fluctuating data stream setting, where data arrival rates oscillate between normal and peak bursts (sometimes fierce), the fact that those figures are unknown a-priori in real-time scenarios could be to blame.

There are QoS metrics that are based on time. For example, throughput and latency.

- **Throughput**. It is loosely defined as the count of streaming tuples that can be processed with specific computation resources during a time period. The goal is normally *high-throughput*. SPEs normally work by implicitly catching up with the oscillation in the data arrival rates aiming to maximize the throughput.
- **Latency**. Is the total time required for processing all tuples arrived during a continuous query (CQ) running session in an end-to-end fashion (i.e., passing through all the operators of a DAG operator graph) from the moment data hits the front-stage of the DSMS coming from a stream ingestion system until results are served to the user, where user chooses to stop the CQ or result outputs to the sink of the data flow graph describing the stream processing operations. The goal of the latency QoS is always *lowering* it.

Another QoS metric depends on the accuracy of results obtained such as:

- **Estimation quality**. If the scenario needs approximation, such as depending on samples instead of the population, error-bound tied to such an approximation determines the estimation quality. *Higher estimation quality* is the goal in this case.

Also, one more QoS metric we consider in this thesis is:

- **Computation resource utilization**. Computation resources are assets. The abundance of extra computing resources does not necessarily mean overprovisioning

them (or under-provisioning them). Those resources are normally shared between various workloads and a QoS aware DSMS should seek to achieve a *high resource utilization*.

Those four QoS metrics are contradicting and solving for all collectively enforces a tradeoff that can be optimized to a specific degree. It worth mentioning though that some DSMSs are working on "best effort" basis where they do not necessarily meet the QoS goals (especially time-based goals), they otherwise work to their maximum capacity trying to achieve as close to the goal specified as possible. Some other DSMSs are designed to guarantee a prespecified set of QoS goals by normally applying cost models so as to reactively (or proactively) guarantee the QoS goals. However, it worth mentioning that current DSMSs are designed to operate in a "best effort" fashion, thus not always being able to guarantee QoS goals specified by the users. A problem that is inflated in spatially-heavy streaming data loads. We otherwise aim at a system that can meet a prespecified list of QoS goals, and also can strike a plausible balance between the contradicting QoS goals. In the next subsection, we explain a general methodology that we apply for measuring the ability of the services we provide in this thesis to meet the QoS goals.

3.2.1 Methodology for Measuring the Achievement of Quality-of-Service Goals

We adopt the following methodology in measuring the ability of each component (i.e., the skill) in achieving a prespecified list of QoS goals. We take a scenario-based methodology. We call our method *cause/effect-tactic-measure*.

The *cause* is the event that causes a QoS issue to arise. The *effect* is the effect of the QoS issue which has happened because of the cause. *Tactics* are the responding mechanisms that we have supported through SpatialDSMS for mitigating the effects (i.e., reversing them). *Measures* are the metrics we impose to measure the ability of every approach (i.e., from the tactics) in achieving the QoS goal.

Categorizing tactics this way allows a more systematic architectural design. Tactic selection decision depends on which way it affects the tradeoff between the participating QoS goals, and also the overall overhead of adopting this technique and whether it is mitigated in a way that renders its adoption beneficial. In other words, the cost of incorporating it does not counteract its benefits. This is because the pattern applied is a trending layered pattern, where

stacked up layers normally add complexity and up-front running costs to the system. Causes include spatial characteristics nature (e.g., skewness, arrival rate fluctuation). Effects include low performance (i.e., in terms of time, throughput, resource utilization, estimation quality). Tactics include element-level optimization, adaptation and approximation. Measures include performance gain, speedup, estimation quality etc. Figure 3.1 shows the workflow of the method.

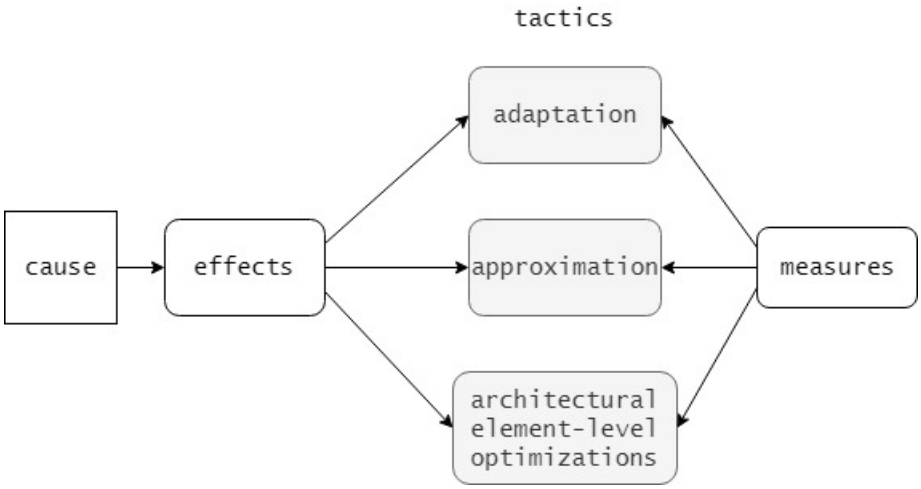

Figure 3.1. cause/effect-tactic-measure for spatially-attuned QoS awareness

3.3 Scalable Storage and Fast Analytics: Better Together

The highly dynamic and scalable application scenarios such as our case scenario (discussed in section 1.1) led to the emergence of our system SpatialDSMS, which resolves the limitations of Lambda architecture, taking advantage of the underlying architecture without its limitations. It is evident that no system can alone survive such highly scalable application scenarios that require a mashup by fusing diverse analytical activities in an interconnected fashion, where the output of a stage feeds another constituent part in an endless fashion, or workloads are continuously mashup. This in its essence means the tight coupling between disk-based storage and online analytics. They collegially complement each other, and real applications need them both in a coherent analytics pipeline.

Two distributed paradigms are gaining more attention. Scalable distributed storage based on NoSQL such as MongoDB [2, 9] and distributed batch and stream processing systems such

as Apache Spark [1]. Current spatial plugins and frameworks that are based on those systems are introduced as marvelous tools that can extract deep insights from massive amounts of digitized spatially-represented datasets. However, they are mostly ad hoc patches and glues that constitute repeated and dispersible efforts that exploit different structures for same targets, causing efforts to fade in a maze of software packages that are hard to consolidate in a coherent structure. On the contrary, a coherent architectural design is needed, which assures that subsequent efforts are conveniently stacked up in a fashion that unifies spatial structures and analytics under one language, aiming ultimately at a robust collaboration for creating new knowledge not inherent in the input spatial sources.

It also worth mentioning that the literature lacks a distributed system for stream interactive spatial analytics (for which we offer SpatialSPEs, the topic of chapter 5).

Mixed workloads identified through empirical investigation show that the notion of "better together" applies in this context and require systems to co-work in a complementary manner. No system can alone handle all types of workloads or keep up with the pace of data fluctuation. Having said that, we posit that integrating features form NoSQL systems with batch processing layers from systems such as Spark and integrating both semantically with speeding analytics services such as Spark Streaming has a sizable impact that achieves better qualities. In the next subsections, we showcase the architectural design of SpatialDSMS.

3.4 SpatialDSMS Overview

In this section we showcase the design process of our system SpatialDSMS, starting by the design goals, and then showing the architecture followed by the scope under which the system operates.

3.4.1 Architectural Design Goals

The architectural design goals we achieve in this thesis are the following.

- **Modularity**. We design a system that is constituting of multiple components operating collaboratively in a way that allows them to be plugged in/out in a hot-swappable fashion (i.e., can be separated and recombined), thus enabling more flexibility as we are considering highly dynamic mixed workloads that require various types of treatment. We have achieved this by implementing our methods

and algorithms as patches or glues that are tightly interfaced and tied to state-of-art de facto standard systems, thus exploiting the underlying functionalities without reinventing the wheel and leaving logistics handling to codebases of the underlying ecosystems (specifically, Apache Spark [1] and MongoDB [2] in this thesis). This was possible because the underlying systems are modular by design and our patches compile down to appropriate abstractions that exploit underlying functionalities without additional efforts.

- **Elasticity**. We offer a variety of operation modes, ranging from exact closed-form solutions to probabilistic approximations, depending on the application scenario aiming at achieving a prespecified list of quality constraints including a tradeoff between the result's accuracy and latency/throughput.
- **Dependability**. We aim at efficiently handling scenarios with oscillating and fluctuating data arrival rates that normally exhibit temporal skewness, specifically for highly dynamic and scalable application scenarios fully loaded with multidimensional geospatially-tagged workloads.
- **Composability**. We aim at offering baselines that can be combined collectively, or in a mashup fashion to solve most interesting highly dynamic application scenarios, rendering them composable from the baseline primitives that we provide.
- **QoS guarantees**. We ensure that the system runs within the boundaries of the specified budget expressed as latency/throughput and accuracy guarantees goals. In addition to maximizing the resource utilization.

More design goals that belong to specific sub-systems are explained in section 4.3.

Aiming at achieving these goals, we have designed SpatialDSMS (short for Spatial Data Stream Management System) that is explained in the next subsection.

3.4.2 SpatialDSMS Architecture

Despite being a promising direction that has been adopted heavily in the literature, because it achieves several QoS goals (i.e., high accuracy, low latency) while operating over massively fast arriving data streams, we posit that Lambda architecture suffers from many limitations that hinders its adoption in geo-referenced fast arriving streaming data loads that

normally exhibit temporal skewness and fluctuation in arrival rates. Perhaps most significantly is the fact that sending same data loads to two distinct storage (i.e., one is batch and the other one could be the fast memory in the speed layer) media overburdens the communication and I/O components. Also, combining current data loads (i.e., streams) with historical archives (i.e., disk-resident), normally using a stream-static join operator has not been addressed. The lack of consolidation and orchestration between the two layer's storage stacks easily causes an overhead that is carried to any custom management effort at the processing layer. Moreover, Lambda architecture is a general architecture that is not attunable with the nature of data that is arriving from heterogeneous data streams, rendering its adoption as-is inappropriate for spatially-laden scenarios. However, Lambda architecture is not a panacea, but otherwise serves as a simple place that has inspired us to design a novel architecture for spatial data stream loads. We aim at enriching such an architecture with QoS-aware optimizations that are attuned to the nature of the arriving data streams (spatial in this case). We aim at transplanting and injecting QoS and spatial awareness transparently within our architecture so that the user in the presentation layer benefits from the overall optimizations provided without the need to reason about the logistics in the underlying layers.

One distinction also departing from Lambda architecture is that in our architecture, in addition to stream (i.e., data-in-motion) data, we allow input data to come as batches (i.e., static, data-at-rest), which could be coming from other sources or legacy systems or even master tables from data lakes or data warehouses. In this way, we guarantee further flexibility that allows systems to operate with ease in highly dynamic environments that request, sometimes unprecedented, mixed workloads.

In this thesis, we have designed an end-to-end QoS-aware data stream management system for the management of mixed workloads of massive amounts of geo-referenced data loads arriving endlessly through heterogeneous fluctuating streaming channels. We dub our system as Spatial Data Streaming Management System (SpatialDSMS hereafter for short).

The context diagram of figure 3.2 depicts a high-level architecture of SpatialDSMS's workflow. Geo-referenced data is coming from heterogeneous sources, to be then served as an unbounded input table (in SPEs terms) at regular basis (e.g., batch intervals, a.k.a. trigger intervals in SPEs terms). At a front-stage resides an interactive interface that confronts users with scalable options depending on the application scenario. For example, the user can either opt for exact processing or approximate processing. Storing data efficiently in a distributed environment before processing it or processing it on-the-fly. All in all, depending on the user's selection, the system takes care of the underlying spatial-aware data management logistics with QoS being natively incorporated, thus not requiring the user to reason about the underlying logistics that are related to the QoS.

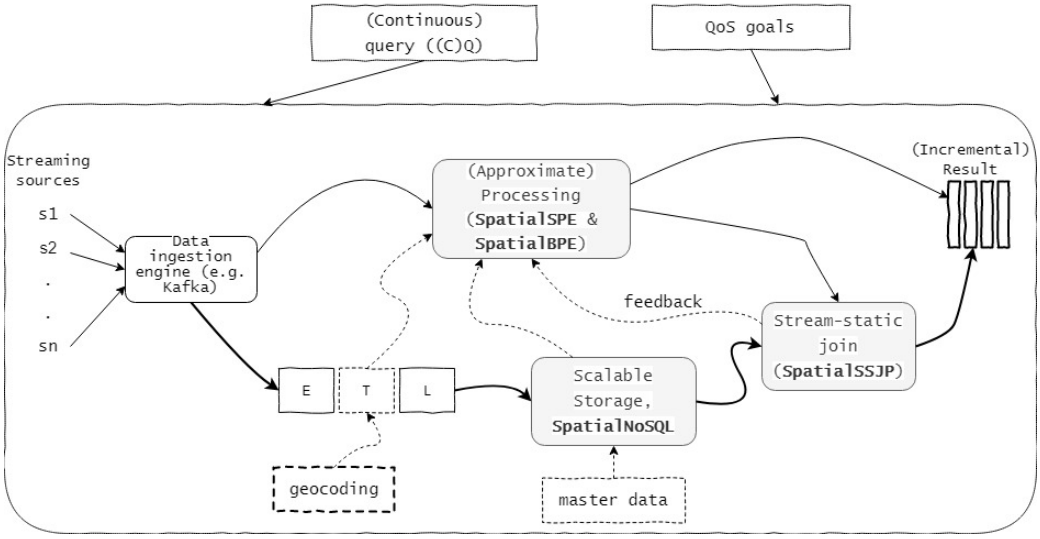

Figure 3.2. SpatialDSMS Overview

All parts constituting our architecture are heavily discussed in their corresponding chapters. On the biggest picture, our architecture resorts to a layer pattern as shown in figure 3.3. Other patterns apply implicitly from the underlying architecture. For example, pipe-and-filter pattern applies from Spark core.

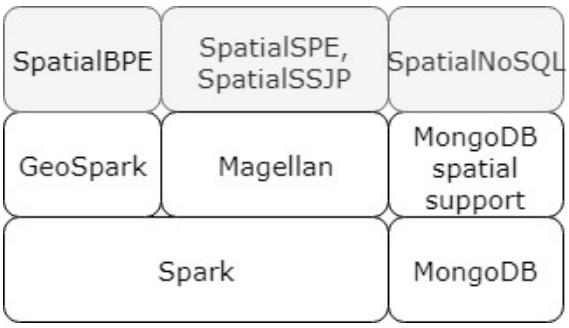

Figure 3.3. layered pattern of SpatialDSMS

In the next subsection, we discuss the presumptions and assumptions under which SpatialDSMS is operating.

3.4.3 Scope of Operation

Many QoS-aware *tactics* exist in the literature, including adaptivity, elasticity and approximation. Those techniques aim mainly at achieving a prespecified list of QoS properties expressed through SLAs. In this thesis, we focus on *adaptivity* and *approximation*. Elastic computing clusters, in which it is more of a default that there are standby computing resources (e.g., executor cores and secondary or main memory resources) to be added dynamically, is outside the scope of this thesis. We only consider parsimonious (stingy or frugal) resource-constrained cloud or in-house cluster computing deployments.

We design our system SpatialDSMS so that it operates under the following assumptions.

- We consider data stream sources with the following characteristics. i) data loads are spatial, which in this thesis is either geometrical planar (i.e., Euclidean plane, flat surface) coordinates (coming, for example, from sensors operating on Global Positioning System, GPS) defined as latitudes and longitudes (i.e., points), or shapes fenced by virtual boundaries representing lines between spatial points, such as polygons. ii) source streams use the push model to push their data to receivers in the ingestion layer, receivers never pull the data. The stream is unbounded and there is no such thing like a “point in time” defining the end of streaming. iii) data is hitting the ingestion receiver only once (i.e., need to be processed in single pass) and data tuples are non-replayable. iv) data loads are temporally oscillating and fluctuating in skewness, meaning that the skewness and size are unpredictable. Because of those

characteristics, Continuous Queries (CQ, a.k.a. online queries) are fundamental, expressed through a Continuous Query Language (CQL).

- Continuous Queries (CQ) are expressed using the fluent API of Spark SQL. They are mapped by the underlying optimizer into a Directed Acyclic Graphs (DAG) consisting of diverse operators (i.e. join) running against micro-batch tuples arriving at each window. Results are incrementalized in the sense that they are improving after each batch interval, which is also known as *incremental stream query evaluation* [42].
- We rely on *tumbling time-based window* semantics, where a sequence of unbounded micro-batches hit an ingestion layer (could be a buffer or a specialized system such as Apache Kafka [43]) with tuples during a time interval (i.e., window). Tumbling windows differ from sliding windows in the sense that they do not overlap.
- We deploy our experiments, for batch and streaming modes, either in a private in-house cluster or a Cloud. We do not put any considerations on the type of processing nodes or memory architectures. High resource utilization is a common QoS goal in server-based and Cloud deployments. We also do not put any considerations on any additional overhead incurred by the communication between the stream sources and the processing infrastructure (i.e., in Cloud or cluster). Further, possible deployments on Fog infrastructures are outside the scope of this thesis.
- The type of parallelization we consider is data parallelization (a.k.a. data-level parallelism or DLP for short, as opposed to task-level parallelism, or TLP for short [44]), where instances of DAGs operators are dispatched to the workers of the computing cluster. Then, a data partitioning scheme is applied to distribute arriving data to the fast memory of the workers, thereafter each operator instance performs a local computation and returns the result to a coordinating node (known as master).
- We only consider optimizing partitioning strategies for the batch processing (as described in chapter 4). We avoid touching those in the streaming analytics as the partitioner can become a bottleneck if the input size is very high. Since we need strategies that can keep up with the pace of burstiness in data loads, the best resort is approximation as it does not require re-partitioning.

- From many types of spatial information, we consider *static spatial points*, *polygons* and *trajectories*. Static points are those points that are incorporated as additional information with a spatial object. Such as locational data that are added as metadata to '*tweets'* in Twitter micro-blogging system, those data do not change over time. On the contrary, trajectories are information generated by objects in-motion, where temporally varying object location is associated with corresponding timestamps, thus constituting an array of static spatial points that collectively form a trajectory. We also consider *static spatial regions* that do not move or evolve over time, such as districts or boroughs in a city (hereafter polygons synonymously).

We believe that those assumption does not affect the generalizability of our systems and algorithms that we present in this thesis. In the next chapter we introduce two sub-systems; SpatialBPE and SpatialNoSQL for distributed batch processing and scalable distributed storage of big spatial data, respectively.

Chapter 4

QoS Aware Distributed Batch Spatial Query Processing

4.1 Introduction

In this chapter, we describe the methods we have designed throughout SpatialDSMS for supporting quality of service goals in batch processing workloads. As the data coming from streaming sources hits the data ingestion engine, the user may opt for , depending on the scenario, storing (part of) the arrived data ‘as-is’ (thus supporting the construction of data lakes) in a persistent storage backend or in an optimized reformatted NoSQL fashion, thus providing more flexibility, interoperability and scalability. Those materialized data snapshots are then used for offline batch processing in support for interactive analytics. For example, prediction models work better in offline mode and there are, as far as we know, no online predictive machine learning or deep learning models that outstrip offline counterparts in terms of accuracy. As such, those analytics are performed offline in batch modes and results support the online part.

This chapter is organized as follows. We first describe data partitioning in distributed systems in § 4.2, this is followed in § 4.3 by explaining three partitioning goals that are most recurrent in the relevant literature. We then in § 4.4 classify the traditional distributed data partitioning methods, focusing on their limitations that led to the emergence of state-of-art spatial partitioning methods discussed in § 4.5. Thereafter, in § 4.6 we show a general view of a spatial data batch processing system , and in § 4.7 we present SpatialBPE that we have designed for spatial batch analytics in big data frameworks, proceeded by storage-oriented counterpart SpatialNoSQL that we have designed for NoSQL scalable storage as discussed in 4.8, both constituting integral parts of SpatialDSMS.

4.2 A Primer on Distributed Data Partitioning

From the many parallelization methods available in distributed computing environments, we focus specifically on data parallelization. It is loosely defined as applying several instances of an operator (from the DAG graph) on several partitions of the input data parallelly, such as each partition is processed by a single instance. Data from sources is first partitioned using a partitioner into several chunks that are disseminated to parallelly connected computing

nodes (i.e., worker nodes). After each operator finishes its designated task (same operation applied by all instances to different data partitions), it emits an output that is collected together with outputs from other operator instances into a coherent piece using a combiner which then outputs the final result to the user or forwards it as an intermediate result to be ingested by other operators downstream (i.e., complementing the DAG). Figure 4.1 depicts a typical architecture of a parallel data management system and the mechanism of data parallelization.

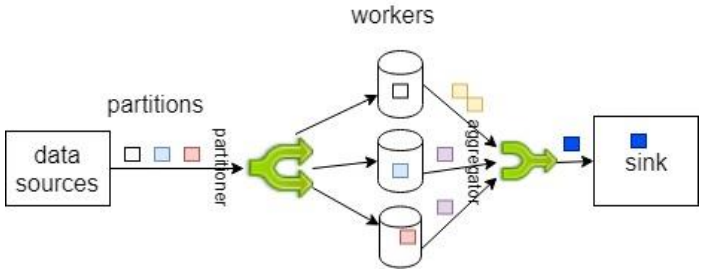

Figure 4.1. An exemplar architecture of a distributed processing system

What is common among all frameworks that apply this model is that there is a splitting (a.k.a. partitioning or sharding) and combining stages. The decision on the mechanism used for splitting is important and may have serious impacts on the overall system performance. We posit that splitting methods should be chosen carefully to increase benefits while minimizing adverse effects. More precisely, it is often the case that dynamic and scalable application scenarios require designing special custom partitioning (splitting) methods that consider, most importantly, the nature of data being treated.

4.3 Spatial Data Partitioning Goals

Current distributed computing systems have one intrinsic problem in common, which is the fact that they are all designed to capture and handle generic workloads, thus are unaware of the nature of data they are handling. Consequently, not being attuned to the data characteristics may degrade significantly the performance achievements to points that sometimes undo the benefits of parallelization.

As the core idea of "bringing computations to data" depends heavily on the fact that "data is appropriately apportioned", which has been mistakenly long explained as a resemblance for "load balancing". Appropriate data splitting does not necessarily imply sending roughly same data loads to every partition in parallel (thus the term 'load balancing'). Our experience with dynamic and scalable application scenarios (such as the envisioned scenario of section 1.1) posits that "load balancing" alone cannot normally achieve the QoS goals desired by the user (as defined in section 3.2).

Most important quality requirements that affect the design of a partitioning strategy include the following:

1) **Interoperability**. In highly dynamic and scalable application scenarios, interoperability plays a pivotal rule in allowing a better consolidation. Data normally originates in heterogeneous sources with different formats and structures. Consolidating all sources under one umbrella that unifies the structure is essential and allows a better interoperability that eases the process of moving data around and consuming it by diverse access tools, and enables a streamlined jumping across the systems that constitute all layers of our architecture. For example, the on-the-fly indexing structure that is used for speeding up the interactive processing of stream data should be the same as the one that is used in the serving and batch layers so that snapshots of stream data seamlessly flow along the pipeline without the need for restructuring. For example, using grid representations with ordering (e.g., z-order curves) structures for both the speed and serving layers (e.g., Spark Streaming and MongoDB, respectively). Additionally, having the same data structures representing different workloads simplifies spatial queries that incorporate a join that is necessary to be performed between different data sets (potentially residing in different storage media). For example, reiterating our case scenario from section 1.1, an interactive query may request "finding all incident locations (regions) where more than 50 volunteers in-motion (dynamic spatial objects) are around within 2 miles ". Using the same representation structure (for example, grid-based imposed with an ordering such as z-curves) for volunteers (as a stream of spatial points) with regions (a static master table, perhaps polygons with same representation structure) can simplify the

stream-static join (a.k.a. geo-fencing, explained in chapter 6 in details) processing by simply overlaying maps (spatial points map and regions map) and the join is done gracefully. The overlay enforces a containment join predicate. Moreover, by achieving this goal, we intuitively avoid repeating the same logic for several workloads and we also avoid the complexities associated with orchestrating the operation of several system units. This also simplifies writing codes that easily jump across operating frameworks of different kinds.

2) **Scalability**. In highly dynamic and scalable application scenarios, arrived data show high temporal skewness and fluctuation. This requires the system to be highly scalable in order to cope up. Stated another way, in distributed data management terms, this means scaling in/out or opting for approximate solutions depending on the scenario. Scaling out seamlessly means provisioning extra processing power, storage capacity and appropriately distributing the data and workloads. The choice should account for repartitioning scenarios, where data need to be repartitioned in case of dynamic allocation (provisioning extra resources or de-provisioning). This normally confluence with the costly challenge of rebalancing partitions, where in such cases the system starts to show undesirable state causing hotspots (i.e., partitions with disproportionate volume of traffic) to appear. Rebalancing simply means migrating data between partitions so as to balance loads. There are two modes of migration, online and offline, where the former allows migrating data while partitions are in-use by some operators, whereas the latter is more disruptive as it requires marking partitions unavailable during migration. Both modes deteriorate the overall QoS goals, mostly rendering the system unable to meet time-based QoS goals such as latency and throughput. Having said that, a successful partitioning strategy should aim at minimizing the shuffling during migration. This can be achieved by better trading off three goals that are described shortly.

We have identified three contradicting goals focusing specifically on spatial data partitioning, which determine the QoS of big spatial query processing. i) *Load balancing*, which is the process of de-clustering data loads in a way that guarantees an even distribution among all partitions, thus mitigating data skewness. While this is efficient for general-

purpose data loads, it is insufficient for geospatial datasets. Spatial data loads often show co-location continuum relations. We refer to this characteristic as ii) *Spatial Data Locality* (SDL) *preservation*. Preserving this co-location feature is essential for an optimized big geospatial data analytics performance. By achieving SDL preservation while splitting data, the partitioning strategy aims at minimizing cross-partition spatial data access operations. For example, proximity-alike spatial queries normally require accessing spatial tuples (representing objects) that are geometrically-nearby. By being able to preserve such a proximity relation while splitting data, by for example sending geographically-nearby objects to same partitions, the system axiomatically reduces cross-partition access as it only accesses some partitions that host appropriate objects. The partitioning scheme should also, for the same purpose and at the same way, aim at minimizing cross-partition joins. iii) *Boundary Spatial Objects* (BSO) minimization. Imagining the earth flattened out (a.k.a. Euclidean space or flat surface) and split into cells (forming a grid network). We refer to spatial objects residing exactly on borders between cells as *Boundary Spatial Objects* (BSO). Accounting for those in a partitioning scheme is specifically challenging, as it imposes extra processing overhead on the system. Specifically, if BSOs constitute a large portion of the spatial dataset. This can be extremely detrimental to the processing operator in cases such as join processing, especially that most well-performing join algorithms are based on *filter-and-refine* approach, where processing BSOs (a.k.a. edge cases) in the refinement stage requires applying the real geometry processor which is computationally expensive and turns prohibitive in extreme scenarios.

An efficient Spatial Data Management Engine (SDME) targets at allocating roughly equal weights of spatial objects to processing elements, preserving, as much as possible, the SDL by grouping geometrically-nearby objects within same subdomains, and minimizing BSOs. To achieve those, various works of the literature have designed spatial-aware custom partitioning strategies that collectively provide top service layer for solving some of those goals in a way that guarantees an acceptable degree of balance between them as discussed hereafter. We evaluate representatives of those works based on the three goals mentioned above. We first review classical data partitioning methods, as complex spatial-aware methods are based on them. Afterwards, we provide taxonomies for spatial-aware partitioning schemes.

4.4 Traditional Big Data Partitioning Schemes

In traditional distributed data management systems, the two most common partitioning strategies are: horizontal and vertical. i) **Horizontal partitioning** (a.k.a. sharding). All partitions (a.k.a. shards) share the same data schema, with each partition hosting a subset of the data. Accessing items horizontally apportioned is more challenging than other schemes, because all partitions share the same schema. This however is amortized normally by designing appropriate access structures (i.e., indexes). In this thesis we focus on horizontal partitioning. ii) **Vertical partitioning**. In columnar databases, each partition host a group of fields (columns in RDBMSs terms). The way division is decided is normally based on the access pattern, such that most frequently accessed fields are placed in a partition while others are hosted in other partitions.

Horizontal data partitioning can be then classified into three schemes as the following: i) **range key-based** data partitioning. It splits tuples based on a specific range of a partitioning key, where each operator instance is assigned non-overlapping key range such that each portion of data that are having that key range are forwarded to the same partition. Initial data assignment tends to be highly imbalanced because some keys are more common than others in real applications. Hence, the selection of the partitioning (a.k.a. sharding) key is pivotal to avoid sequential query scans, by only visiting some partitions for a query. This method is better suited for stateless operators if applied in interactive stream processing systems (i.e., not appropriate for online aggregations which are stateful operations). ii) **hash data partitioning**. It employs a hash function to partition data, where same-group data share the same hash value (i.e., hash range). In this case, depending on the application and the adopted hash function, data locality might not be well preserved. Also, initial data distribution tends to be imbalanced. If hash function is selected appropriately, hash benefits the parallel hash join immensely. iii) **Random and round-robin** data partitioning. It partitions data based on a given equation, where every tuple in turn is assigned to a partition randomly or sequentially (for example, in a clockwise direction). The merit of this method is an even data loads distribution [45] . However, SDL is lost, and a query search needs to scan all partitions [46]. If used online, this partitioner is not suitable for online aggregations (i.e., stateful), it is only used for stateless operators, that process data chunks independently.

Those schemes are explained as if they were operating on batch workloads that need to be processed offline. However, the story in interactive settings is different. SPEs use a combination of those batch partitioning models with other semantics to split data continuously as they arrive. Micro-batching models do not follow the same partitioning schemes as record-at-a-time models. The former treats the streaming loads pretty much the same way as if they were batches that are processed statically, hence the term "micro-batch". For example, in Spark Streaming, a receiver accumulates stream data at every time interval (i.e., batch interval) into micro-batches (e.g., small RDDs in Spark terms) using a block manager (i.e., technological block in Spark core) and then partitions every micro-batch the same way as if it was a static load, using the default partitioner or a user-preferred partitioner.

Non-relational systems, MongoDB natively supports range and hash data partitioning. The default is range data partitioning, where nearby documents (analogues to tuples in RDBMSs) with close key values are placed on the same partition (i.e., shard in MongoDB terms). Being column-oriented databases, HBase and Cassandra apply vertical partitioning approaches.

It worth noticing that partitioning in processing-oriented ecosystems is not profoundly different than that of storage-oriented systems. However, partitioning is performed in-memory after the *Map* stage and just before the *Reduce* stage in an ad-hoc style (i.e., on-the-fly), where each data partition is passed to a single *Reduce* stage. Spark default partitioning scheme is a hash data partitioner. It also uses range data partitioning among others [47] . Spark provides a mechanism to custom data partitioning for performance tuning in specific application domains. For batch-oriented systems, Hadoop [10] supports hash data partitioning and others.

As noticed, round-robin is not widely accepted by big data distributed management systems. The reason is that although it ensures load balance, data locality is not well preserved, and any query naively scans all partitions. In other words, there is no chance to apply aggressive pruning (i.e., where only specific partitions are scanned). On the flipside, hash and range partitioning strategies preserves locality better than that of round-robin, thus widely accepted. Round-robin, hash, and range are one-dimensional partitioning methods, rendering their 'as-is' application to multi-dimensional spatial datasets inconvenient. Therefore, spatial partitioning methods are required, which is the central discussion of the next subsection. We

first recapitulate data structures that support spatial data partitioning, thereafter we briefly review trending spatially-attuned data splitting schemes.

4.5 Spatial-aware Distributed Data Partitioning

Traditional data partitioning methods were designed for general data structures. However, they are unaware of specific characteristics of spatial data. We first review the most common spatial partitioning approaches and build taxonomies for their application in modern big data management systems. Afterwards, we identify their pros and cons regarding the three goals mentioned in section 4.3.

4.5.1 Multidimensional Data Structures Supporting Spatial Data Partitioning

First, we summarize multidimensional data structures that support spatial data partitioning. Performing spatial analytics on highly dynamic big data streams require two stages, space representation (two alternatives are common, data-dependent or independent) and access data structures. The former step is representing the space where points are drawn from using a data structure such as grids, whereas the latter is responsible for selecting appropriate data indexing structures (a.k.a. access structures) for speeding up the access at query run-time. We focus on indexing structures that are used for both spatial points and shapes (such as polygons). Space representation implies a division structure that is either data-dependent or space-dependent [48]. Stated another way, space representation is followed by imposing a data access structure on the representation for speeding up the scans. There are two types of spatial representations, deterministic and probabilistic.

One of the most widely used and accepted deterministic representation structures are hierarchical representations such as grid-based and tree-based structures

1) **Grid-based** representation structures [49, 50] . As its name implies, in two dimensional spaces (imagining the earth flattened out) it partitions the embedding space (the geometric space where geospatial data resides) into grid-shape (rectangular or squared) cells by placing a grid over it (i.e., overlaying it). A pointer is referencing a data structure (e.g., an array) that hosts the real elements (spatial objects) of a specific cell.
2) **Tree-based** representation structures. An example in this category is quadtrees [51] and k-d trees [52]. They basically work by dividing a two-dimensional

planar geometry (i.e., Euclidean) recursively into four rectangular parts. The only way to return a query result from data distributed with a tree structure is to traverse tree nodes, this implies that an optimization should seek minimizing, as much as possible, the visited nodes during run time.

3) **Ordering-based** representation-enriching structures. An ordering is normally assigned to cells in a grid decomposition, and thereafter a tree-based access structure (e.g., B+-tree) is imposed on the ordering. Ordering is a multidimensional reduction approach that projects multidimensional cells into a one-dimensional space. From many types of ordering we focus on the family of z-order curves (a.k.a. Morton orders). Simply put, an ordering is an enhanced representation employed after other representations. It worth mentioning that ordering per se is not a representation structure that can be used alone, it lends itself otherwise as a helper that can enhance preceding representations so that the access is sped up. Ordering helps in deciding the traversal order of grid cells.

Variations to Morton ordering include methods that map the grid into geo*codes* (for example, geohash). Those codes if sorted (e.g., in an ascending order) results in an ordering that is equal to the order of visiting leaf nodes (i.e., representing grid cells) of a tree (e.g., quadtree) representation.

A special application of Z-order curves is geohash [3], where the ordering imposed on the grid space is z-shaped, geocodes generated are strings where a shared prefix signifies geometrically-nearby spatial points, where longer shared prefix means objects involved are closer in real geometries. Geohashing is exemplar in quick-and-dirty proximity searches (i.e., working as a quick-and-dirty sieve).

After representing the spatial object (being point, region, etc.,), an access structure (i.e., index) is imposed on the representation to speed up spatial search queries (e.g., range). Most common access structures that are associated with hierarchical representations include

[3] http://geohash.org/

arrays, where each element of the array references a cell in the grid representation. Also, tree indices (such as B+-trees and PK-tree [53] , both can be imposed on a quadtree representation), where each leaf node references a cell.

Hierarchical representations are not appropriate for interactive settings, where massive data arrives very fast with fluctuation in skewness. The reason is that the creation and management of those structures is expensive. Alternative solutions comprise the exploitation of approximate structures, such as Minimum Bounding Rectangles (MBR). Methods include tree-based representations such as R-tree [54, 55] . R-tree works by grouping objects based on their proximity (their enclosing MBRs specifically). An inherent problem is that point queries are costly and may need to visit unduly all the nodes because of the overlapping nature of the MBRs. R-trees are widely used in online steam settings because of the dynamicity they provide in such a way that they do not require fully reconstructing a tree upon receiving a new stream tuple and it can otherwise be placed in a hot-swappable fashion. R+-tree [56] differs from R-tree in that it generates non-overlapping MBRs. It does so by dividing the space in non-overlapping regions, and a spatial object may span multiple regions (B-tree can be used to group those regions).

Grid approaches and those based on quadtree has a paramount utility in operations that require datasets mashup and the incorporation of diverse operations between batch and interactive modes of operation. It is then evident that those approaches perform well in stream-static join (more details in chapter 6), an advantage that is further inflated when enriching them with ordering sequences such as Z-order curves. Also, recent studies [57, 58] have shown that using non-hierarchical and simple spatial indexes on modern parallel systems boosts the analytical performance. However, an inherent problem in grid partitioning is that it exaggerates the load-balancing problem in dynamic application scenarios, where specific cells are easily becoming stragglers (i.e. congested) while others are empty. Stated another way, fixed grid partitioning is not preferred in highly skewed distributions, instead the grid size should be based on a cost model that considers data distribution and BSOs, so as to preserve spatial proximity and minimize BSOs in addition to load balancing.

Because every representation method (i.e., partitioning in the context of this thesis) has its own limitations. We posit that applying a single scheme cannot guarantee the QoS goals and

is not able to achieve the three spatial partitioning goals that we have identified in section 4.3. Consequently, most works of the relevant literature have designed their own set of custom spatial-aware partitioning methods that are based on the primitives discussed in this subsection.

4.5.2 Custom Spatial-Aware Data Partitioning methods

Custom spatial-aware data partitioning methods that fall within the confluence of many schemes mentioned in section 4.5.1 include:

I) Sort-Tile Partition (STP) [59] . First, data sorting is performed in one dimension (horizontal in a grid-based Euclidean representation), and equally-loaded slices are generated, thereafter data in each slice is sorted and partitioned based on the other dimension (vertical in a grid-based flat surface representation). Tiles (horizontal and vertical) locations can be selected based on a model that balances the tradeoff of the three goals of section 4.3.

II) Boundary Optimized Strip Partitioning [60]. This algorithm is specifically designed for BSO minimization. It is a special case of strip (tile) partitioning where a cost model is applied to select optimized tile locations so as to minimize the BSOs.

III) Custom partitioning methods. For example, [61] have designed a boundary-aware spatial splitting scheme that also achieves load balancing. Also, Cruncher [62] employs a dynamic adaptive method that is aware of query workload. A cost-model-based repartitioning module calculates number of points and queries for each partition and repartitions accordingly.

SpatialHadoop [63] supports grid-based, sometimes enriching with an ordering such as Hilbert- and z-order curves, STP, and quadtree [64] . HadoopGIS currently supports grid-based partitioning in addition to others added through SATO framework [65] . From the in-memory batch processing systems, SpatialSpark supports grid-based and STP [66] . On the other hand, GeoSpark [11] supports grid-based splitting, sometimes enriching by Hilbert-curves ordering, in addition to quadtree, Voronoi and R-tree. MongoDB does not support partitioning on geospatial keys.

Intensifying now on the ability of each method in meeting spatial partitioning goals (recap section 4.3), load balancing, BSOs minimization and SDL preservation. We now provide a guidance taxonomy that facilitates the selection of most appropriate schemes for specific domains. For example, considering spatial data skewness ratios for co-location data mining promotes selecting a framework that employs a locality-aware custom partitioning method. In grid-based partitioning, the selection of the partition size profoundly impacts the performance. For example, a coarser-level amplifies data imbalance, where some partitions may acquire more elements than others. By way of contrast, a granular-level improves load balancing, however, amplifies BSOs. This method replicates BSOs to neighboring grid cells. However, it does not provide a capability for achieving SDL preservation goal. Quadtree-based schemes can handle BSOs by replicating them to adjacent overlapping cells, and its ability to balance loads depends on data distribution in real geometries. Also, it does not support SDL preservation. In STP, BSOs are processed by expanding neighboring cells. Also, STP guarantees load balancing to some extent by applying a splitting mechanism that is aware of actual data distribution. However, it does not intrinsically achieve SDL preservation. Imposing an ordering (e.g., z-curves) over the representation helps in preserving SDL. Table 4.1 sums-up our taxonomy, comparing the performance of the spatial partitioning techniques sketched previously, and introducing an important dimension that shows capability of every method in achieving the three partitioning goals.

None of those schemes efficiently imposes a balanced tradeoff between the three spatial-aware partitioning goals. Those primitive schemes insufficiently represent spatial objects relationships for specific application scenarios, such as smart cities, by being not attuned to spatial locality characteristic. As a resolution for this, some works have gone beyond those traditional spatial partitioning methods by designing custom partitioning schemes that strike a balance between the three partitioning goals. For example, [67] have designed a query-workload-aware technique for partitioning big spatial data that adaptively repartitions data in accordance with a query workload, achieving roughly equal load balances while keeping SDL preservation in check.

It is then evident that “one fits all” does not apply when it comes to selecting a spatial data partitioning approach that balances the three goals, SDL preservation, LB and BSO

minimization. Stated another way, we are not aware of any single method that alone successfully tackled the problem holistically. Hence, a better-performing method should be custom, adaptive or both combining many primitives in a coherent way taking advantage of the overarching traits of each method individually in a way which guarantees that they reinforce each other without their limitations.

Table 4.1. A taxonomy of capabilities of general spatial splitting methods in handling spatial partitioning goals defined in section 4.3

Approach	**big spatial data partitioning goals**		
	load balancing	BSO Minimization	SDL preservation
Grid-based	✓	X	X
Quadtree	✓	X	X
STP-based	✓	✓	✓
Grid with space-ordering	✓	✓	✓

Notice that z-curves and a method based on STP are the only two amid all others that can achieve a weighted balance between the three tradeoffs discussed in section 4.3. This rationale our selection for those two in designing QoS-aware partitioning strategies in support for distributed systems serving highly scalable application scenarios (NoSQL or batch processing frameworks). We have designed hybrid adaptive data partitioning methods as we posit that no single method alone can balance the tradeoffs between the three partitioning goals (section 4.3).

As a recap, referring to the schematic diagram of figure 3.2, The user may opt for storing the data coming from the streaming sources as-is, in which case the stored data is not distributed, it is otherwise stored as a whole chunk. In a later stage, that data can then be distributed for parallel processing (for example, using Spark). For this case we have designed a novel adaptive spatial-aware partitioning method (discussed in section 4.7.3.1) that better trades-

off the three goals mentioned in section 4.3 in a fashion that achieve results outperforming state-of-art methods.

On the other side, for scenarios where data arriving is tremendous and cannot fit efficiently in single chunks, simply because it is arriving from diverse heterogeneous sources, the system may opt for unifying the format (thus supporting the interoperability requirement) and distributing then the arriving data to be stored in parallel storage chunks (shards in MongoDB terms). For an efficient dissemination of spatial data, aiming at a weighted QoS-aware tradeoff between load balancing and spatial locality preservation, we have designed a qualified partitioning strategy for NoSQL distributed environments (discussed shortly in section 4.8).

4.6 System Design Perspectives

We have designed two sub-systems. One for batch processing that we dub as SpatialBPE, and one for NoSQL scalable storage that we term as SpatialNoSQL.

By this combination, we recap our SpatialDSMS, where we have two components (analogous to batch and serving layers of Lambda architecture). Any snapshot or view (resulting from online processing or batch processing) or simply pouring as raw data directly heading towards the storage backend will be indexed with an appropriate representation structure, and thereby will be partitioned in a unified manner so that future workloads mashup seamlessly. For example, we use the same indexing strategy (grid with an enforced z-order curves) to index streaming data coming for an online processing, and also to shard the data in NoSQL (i.e. MongoDB). In addition, we exploit the same dimensionality reduction approach for an offline batch processing of data (e.g., Spark). This structure streamlines the mix workload handling which is a main design goal of SpatialDSMS. In the next subsection, we explain SpatialBPE.

4.7 SpatialBPE: Spatial-aware Batch Processing Engine

SpatialBPE[4] constitutes two main submodules: spatial-aware partitioning module and query optimizer module, which are explained in the following two subsections, respectively.

4.7.1 Motivation

Big data is being exploited in various emerging scenarios that are dynamic and require high scalable architectures. For example, participatory healthcare services [68] , city planning [69] and urban computing [5]. Batch-processing systems are not designed to process that data deluge. This led to the appearance of parallel computing frameworks such as MapReduce-based [70] systems and NoSQL scalable storage systems such as MongoDB [2]. Two entwined aspects apparently intermix in dynamic application workloads. Those are data splitting and query processing. Load balancing while partitioning data has been on full display by those systems in the relevant literature, not considering the spatial characteristics, such as the data skewness, where geo-referenced objects concentrate at some locations more than others in real geometries [71, 72] . In addition, boundary spatial objects (BSO) minimization goal has not been adequately considered, thus deteriorating the benefits of parallelization, recap that BSOs are objects that reside exactly on the borders between grid cells (in grid-based hierarchical representations). This carries a negative impact that renders the underlying system unable to handle QoS objectives in complex scenarios such as geo-clustering algorithms. Works of the literature have focused on a *replicate-and-refinement approach*, where BSOs are duplicated to adjacent cells, which is followed by a refinement step [65], taking a huge toll that renders the system unable to meet time-based and accuracy QoS goals. It has statistically been proved that geometrically proximate units share similar characteristics by being affected by the same surrounding factors (such as ecological factors in environment monitoring studies). This implies that the system should be attuned to spatial co-location relationships while partitioning data, thus focusing on SDL preservation has a paramount effect on performance on the way to achieve time and accuracy based QoS goals.

[4] The source code of SpatialBPE (including SCAP) is available at: https://github.com/IsamAljawarneh/SpatialBPE

Most scenarios in dynamic environments seek answers (through spatial queries) that reflect a proximity and co-location relationship. We posit that preserving SDL leads to avoiding costly shuffling. Cross-nodes shuffling is known otherwise to heavily cause the processing system to run into a big bill far beyond its capacities.

However, current batch processing systems do not natively offer appropriate approaches for efficiently trading-off partitioning goals in an aim at achieving QoS goals. To overcome those limitations, we have designed and implemented an in-memory batch processing component for SpatialDSMS. The contributions of this component are two folds. First, we design a custom spatially-attuned partitioning method that achieves a plausible degree of load balancing in addition to BSO minimization and/or (depending on the case scenario) SDL preservation. Thereafter, we design query optimizers that appropriately exploit the newly added partitioning method in solving density-based clustering algorithms (specifically DBSCAN-MR) with prespecified sets of QoS guarantees.

4.7.2 Design Perspectives

Figure 4.2 shows a high-level architecture of our spatial-aware optimizations for the in-memory spatial batch processing systems. Our patches reside atop Spark's Magellan (which itself sits atop the core of Spark) constituting a transparent layer that hides implementation details from application layer, thus achieving one of the design goals of SpatialDSMS, which is the '*modularity'* (refer to section 3.4.1 for details).

Our patches include a spatial-aware partitioning scheme (we dub as SCAP, explained shortly in section 4.7.3.1), basically accounting for a better tradeoff between three goals (load balancing, BSOs minimization and SDL preservation), aiming ultimately at achieving a weighted balanced tradeoff between the QoS contradicting goals: low-latency, high-throughput, high-accuracy and high resource utilization.

An integral part of SpatialBPE is a query optimizer that employs SCAP and a retrofitted version of DBSCAN-MR [38] implemented over Spark's Magellan for optimizing the parallel execution of DBSCAN [37]. We have specifically selected clustering as an analytics to support for in-memory systems because it is heavily appearing in dynamic application scenarios (refer to section 1.1 for more details).

4.7.3 Spatial Partitioning in Distributed Batch In-memory Processing Systems

Partitioning per se is not an optimization goal, it is otherwise a mean-to-an-end. The goal then is exploiting a well-performing partitioning scheme in analytics and achieving desirable set of QoS goals predefined in SLAs. In this thesis, as batch in-memory distributed data processing systems are responsible for handling heavy workloads in dynamic application scenarios that require scalability, we have designed a spatial aware adaptive big data partitioning method that significantly outperforms baselines by orders of magnitude. Our method is explained in the next subsection.

4.7.3.1 Spatial Co-Locality-Aware Partitioner (SCAP)

By designing a custom spatial-aware partitioning scheme for batch processing systems we focus on time-based QoS goals including low-latency/high-throughput and other qualities such as accuracy and high-resource utilization.

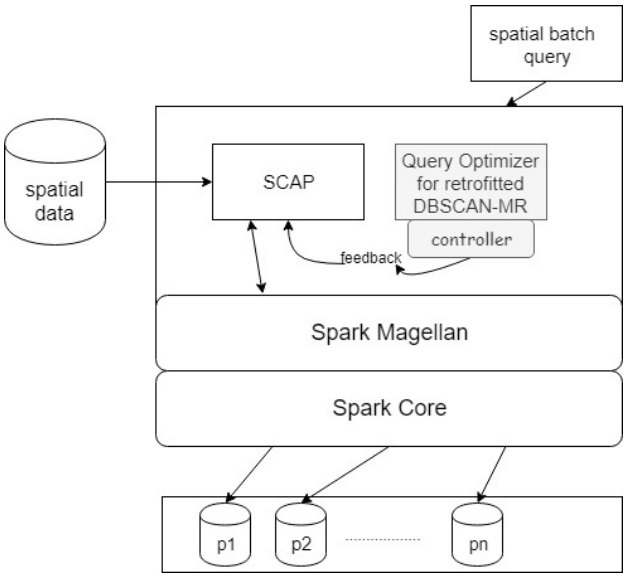

Figure 4.2. SpatialBPE overview

Several works of the relevant literature apply grid-based representations for partitioning spatial big datasets in parallel batch computing settings. However, the plain application of those hierarchical schemes leaves the computing cluster lopsided, where more objects are clumped into few partitions, deteriorating the load balancing. More importantly, spatial co-

locality is impeded as geometrically-collocated objects are randomly forwarded to different partitions, forcing a huge toll caused by the great amount of shuffling that may be required at query time. To alleviate those problems, we have designed an adaptive spatially-attuned

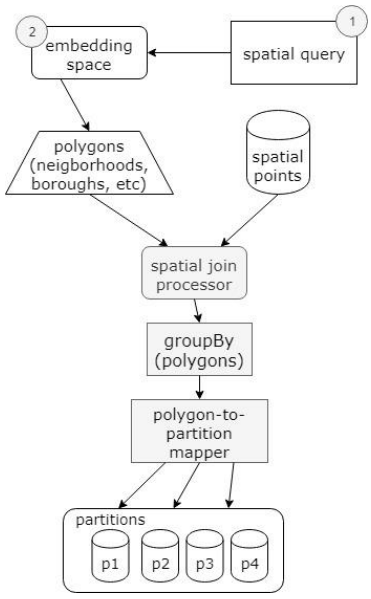

Figure 4.3. Spatial Co-Locality-aware partitioner (SCAP)

partitioning method that considers, most importantly, the SDL preservation and aims at achieving a weighted balance for the other two partitioning goals (BSOs to a lesser extent and load balancing). We dub our novel method as SCAP (short for spatial co-locality-aware partitioning) as shown in figure 4.3.

More formally, the workflow of our method is listed in Algorithm 4.1. The method starts by geocoding the spatial points. We focus in this thesis on dimensionality reduction based on geohash encoding. We then apply an efficient spatial join method that is readily offered by Spark's Magellan for joining the spatial objects (i.e., points) with a table comprising neighborhoods (in city management terms) represented as polygons. This stage results in a list that specifies to which neighborhood each point (i.e., spatial object) belongs. This process is also known as *geofencing*, a problem that demands solving a mathematically resource-extravagant operation known as point-in-polygon (PIP for short). Nevertheless, by employing the already optimized Spark's Magellan library, we significantly cut off the computational costs to linearly discernible margins. Stated another way, this stage resembles

clumping geometrically co-located spatial objects into single patches that can afterwards be disseminated to nearby or same partitions for local processing. For each neighborhood list, we verify whether the number of enclosed objects exceeds a prespecified threshold, which then signifies the need to sub-split that overloaded segment so that to achieve a credible degree of load balancing.

Algorithm 4.1 SCAP partitioning scheme for in-memory batch processing frameworks

```
/* points: coordinates in longitude/latitude shape, neighborhoods: polygonal shapes representing
neighbourhoods of the embedding study area, geoPrec: geohash precision */
Input: points, neighbourhoods, geoPrec, spatialQuery, numPartitions
pointsAssignedList = [ ] //each element contains all points that belongs to a specific neighborhood
finalist = [ ] /* the final list containing sub-lists, where each element (sub-list) will be sent to a single
partition */
coverGeo ← getCoverGeo (neighbourhoods, geoPrec) /* List of geohashes covering each
neighbourhood (polygon)*/
GeoCodedPoints ← geoEncode(points)
/* perform inner join on geohash using the filter stage, filter-and-refine approach */
/* pointsAssignedList: list of points that have been assigned to neighborhoods */
pointsAssignedList = GeoCodedPoints.join(coverGeo, GeoCodedPoints ("index") ==
coverGeo("index"))
/* iterating through all parent lists, where each list contains elements belonging to a single
neighborhood */
For idx = 0 until pointsAssignedList.size
//check whether a number of elements in a specific neighborhood exceeds a threshold
//currently threshold is specified by pre-profiling the data
    If (pointsAssignedList[idx]. count > threshold)
        sub_ pointsAssignedList = split (pointsAssignedList[idx])
        finalist.append (sub_ pointsAssignedList)
    Else
        finalist.append (pointsAssignedList[idx])
    End if
End For
For j = 0 until finalist.size
//materializing data chunks in partitions
    partition[j]. populate (finalist[j])
End for
```

We achieve that by splitting overloaded neighborhoods to granular sub-lists so that the number of enclosed points fenced within each sub-list never exceeds the fences of the prespecified threshold. We do this to account for load balancing, where we seek trading that off with spatial col-locality preservation. To a lesser extent, our SCAP method accounts to BSO minimization, which is attainable through tweaking geohash precision parameter. The workflow of SCAP is schematically shown in figure 4.3.

Procedure 4.1: split (pointsAssignedList[idx], threshold)

```
// the purpose of this method is to split overloaded lists depending on a prespecified threshold
size ← pointsAssignedList[idx]. size( ) //overloaded list size
sub_ pointsAssignedList = [ ] /* each element of this array contains part of the spatial points from an overloaded parent list (where the parent list represents a full neighborhood, whereas the child lists represent parts of the neighborhood) */
newListsCount ← (size / threshold) //the number of new sub-lists (child lists)
index = 0
For i = 0 until newListsCount
    partialList = pointsAssignedList[idx]. take ("*"). where (id between index and (threshold+index-1))
    sub_ pointsAssignedList[i]. append (partialList)
    index = index + threshold
End
Return sub_ pointsAssignedList
```

In the next subsection, we show how we have retrofitted SCAP (with a very little effort) and applied it to Magellan-based DBSCAN-MR, thus retrofitting the latter and gluing it within the layers of Spark's Magellan, which then constitutes a primary contribution of SpatialBPE.

4.7.4 A Recap on Spatial Querying in Batch Oriented Systems

Partitioning geo-referenced big data in parallel computing environments is a precursor for optimizations that aim at achieving QoS goals by applying spatial analytics. Current plain systems being not attuned to the spatial patchy distributions are causing the underlying optimizers to shuffle huge amounts of data over the network. This normally deteriorates the benefits of parallelization, especially in cases where shuffled subsets constitute big fractions

of data. Query optimizers aim at selecting the most efficient query plan that reduces shuffling and thereby striking a weighted balance between the QoS goals.

Having designed SCAP, we have decided to proceed it with designing a query optimizer that exploits its overarching traits in optimizing the running of a costly density-based clustering algorithm (i.e., DBSCAN [37]) that is very common in dynamic scenarios of smart cities. In a loose way, DBSCAN clusters units in a way that considers high-dense regions as clusters, whereas others are noise. The plain DBSCAN is not applicable per se in distributed systems, then DBSCAN-MR [73] has emerged as a variation that is able to run in parallel. Technically speaking, DBSCAN-MR proceeds as follows. It receives *epsilon* (ε) and *minPoints* (short for minimum number of points) as input parameters. *Epsilon* is used to find all points that are far-away from a query point (similar to *k*NN) by a distance that equals *epsilon* at most. *minPoints* is thus the minimum count of points in vicinity that together form a cluster. It then starts by splitting input data points to the partitions of the worker nodes that are comprising a distributed computing cluster. In a later stage, the algorithm employs a local edition of the vanilla DBSCAN for data that is fenced in each partition. The algorithm afterwards merges micro-clusters received from local versions into unified macro-clusters. An apparent obstacle while parallelizing DBSCAN-MR version is the demand to duplicate BSOs into adjoining cells. Considering a planar earth geometry, those cells resemble grid cells that result from the grid representation of the embedding space. In that sense, dealing with BSOs resembles a replicate-and-refine approach, where duplicated BSOs are thereafter eliminated in a post-replication refinement step. Another confounding challenge lies in finding an appropriate manner to efficiently consider SDL preservation throughout partitioning and also striking a credible balance among the three partitioning goals mentioned in section 4.3.

Out of the box, Spark's Magellan does not offer over-the-counter DBSCAN-MR optimizers. We then offer an optimization to Spark's Magellan that transparently incorporate our retrofitted version of DBSCAN-MR, which then constitutes one of our contributions in SpatialBPE. In addition, our retrofitted version of DBSCAN-MR exploits our SCAP method (see subsection 4.7.3.1 for details) for orders of magnitude performance gain (as opposed to the plain DBSCAN-MR) achieving a better balance between QoS goals (especially low-

latency/high-throughput, high-resource utilization and high accuracy) by consequently striking a plausible balance between the spatial partitioning goals.

4.7.5 Spatial Query Optimizers for Distributed Data Batch Processing

4.7.5.1 Co-location Query Optimizer

We have hybridized a retrofitted version of SCAP with a retrofitted version of the plain DBSCAN-MR. In this way, SCAP acts as a front-stage that partitions the static (i.e., disk-resident) input data into the computing cluster, thereafter the retrofitted version of DBSCAN-MR works on the apportioned data as explained shortly.

To be able to apply SCAP to DBSCAN-MR, we have retrofitted SCAP so that it accounts for the BSOs. The approach we choose for dealing with the BSOs is replicate-and-refine. We first replicate BSOs to the overlapping cells. In a later stage, we discard those local duplicated BSOs from the resulting final clusters (a.k.a. macro-clusters). The replication strategy relies on a fact that each neighborhood (i.e., polygon) is represented by several geohashes. In this context, a geohash spans many neighborhoods. As the time of this writing, to avoid introducing any additional model-based cost layers that may bog down the system, we simply replicate objects which belong to overlapping geohashes. The number of replicated BSOs relies on two factors, the data skewness and the geohash precision. Geohash precision is a number that is a multiple of five. As an example, a geohash precision that equals 30 signifies that the size of the corresponding geohash string would be six, which is typically a combination of characters and numbers, whereas a geohash precision 35 means that the string size is seven and so on.

In our previous works *[74, 75]*, we have designed SASAP (explained shortly in more details in section 4.7.5.2), which is a method that is similar to STP (recap information from section 4.5.2). In SASAP, for replicating BSOs, we rely on stretching each strip (i.e., horizontal or vertical split) to a dimension that equals to a double *epsilon* value. We have exploited *Haversine* formula for measuring distances horizontally and vertically. Our SASAP method in our previous work resembles an STP approach, which then requires sorting geospatial points in each direction based on longitude and latitude data. After deeply investigating the possible consequences, we have discovered that the computational model-based approach

that we have applied for computing BSOs to replicate could easily turn as a bottleneck when the data size increases. This potentially can bog down the system performance and, in some cases, may bring the system into a halt. This in part is due to the fact that the procedure sorts massive amounts of multidimensional spatial data (i.e., longitude and latitude). As a way of contrast, our novel method SCAP avoids any complexity layer that necessitates expensive model-based computations for deciding the BSOs to replicate. We otherwise rely on a dimensionality reduction approach that is based on z-order curves, specifically geohash spatial encoding, which reduces the problem of working with multi-dimensionally-shaped data down to that of a single-dimension.

In both methods, SCAP and SASAP, grid cells are overlapping, and the splitting is non-disjoint. While each non-boundary point is assigned a unique identifier, BSOs are assigned several identifiers (one corresponding to each partition they are replicated in). The algorithm proceeds then by sending points to corresponding partitions. A local plain DBSCAN is thereafter applied to each partition. The algorithm now proceeds normally as in the plain DBSCAN-MR. Despite the ability of SCAP (and also SASAP form our previous works [74, 75],) in reducing the shuffling during the local application of DBSCAN in each partition, it induces a huge toll on resource utilization, and here is where the adaptive controller comes into play, lending itself as a loop feedback mechanism from the control theory, aiming basically at minimizing the BSOs but at the same time balancing loads and preserving SDL in a plausibly significant weighted balance fashion. For SCAP, we simply depend on the fact that the geohash precision is tweakable, thus allowing the opportunity for better resource utilization. Interested readers are referred to our previous works for more information on the mechanism at which our traditional method SASAP is acting adaptively. See Appendix B for technical details on DBSCAN-MR.

4.7.5.2 Usage Model and Baseline System

Referring to our scenario in section 1.1, the application may need to build clustering views (i.e., offline) based on locational data of passing-by registered people (e.g., volunteers) who are capable and willing to provide instant assistance to victims of an incident. Those views need to be updated regularly, and this step can be considered as a second stage for the online

clustering, or what is better referred to as macro-clustering [76] (refer to section 7.2 for details) that normally runs offline (e.g., overnight).

Since we have already designed an optimized custom spatial partitioning method in a previous work [74, 75], which we dubbed as SASAP (short for Spatial Aware Self-Adaptive Partitioning). We have decided in this thesis to select SASAP as a baseline benchmark to compare with our newly introduced method, SCAP. SASAP in its core recovers an adaptive-STP-alike (recap information from section 4.5.2) partitioning approach. The working mechanism of SASAP proceeds as follows. First, it accepts spatial data points as an RDD (in Spark terms). Thereafter, it exploits an abstraction from GeoSpark to transforms those into a pointRDD representation (which is an abstraction for spatial representations in GeoSpark that resembles the traditional RDDs from Spark but instead is intended to spatial settings). As it reaches this stage, points are imagined as if they were overlay on a planar earth geometry that is a flattened version of the Earth. Afterwards, SASAP sorts pointRDD points in both directions (i.e., longitude and latitude) and then assigns a unique numerical identifier to every point in pointRDD. SASAP then proceeds by employing a splitting mechanism as follows. First, it overlay dividing vertical stripes on the two-dimensional map that is representing the embedding Earth planar geometry. The result of this stage is a list of longitudes for the stripes. This is proceeded this by a horizontal division for obtaining latitudes of stripes. The overall result of this splitting scheme is a grid in two-directional sorted order. This procedure recovers recursive halving in single-dimension (for each splitting direction) and quartering in two-dimensions (while extracting longitudes and latitudes of stripes).

SASAP is a generic approach that we could apply to various dynamic workloads. A significant disadvantage of SASAP however is that it resorts to an STP approach (recap information from section 4.5.2). We have proved practically its superiority over traditional benchmarks (refer to our papers for interesting results [74, 75]) in accomplishing discernible balance between several challenging spatial data partitioning goals, such as SDL preservation and BSOs minimization. However, we find that an undesirable overhead can significantly accumulate as the data size becomes massively big. Overall, this may deteriorate at some points the benefits we reap from parallelization. This has to do with the fact that several expensive model-based computations are involved. For this reason, we have

decided to design a new method in this thesis that we term as SCAP so that it overcomes the drawbacks of SASAP.

4.7.5.3 Experimental Setup and Test Cases

This section discusses deployment settings that aim at assessing the ability of SCAP and the query optimizer, consequently, in achieving a plausible balance between QoS goals of this thesis.

Deployment and experimental settings. We run our system, SpatialBPE, on a Microsoft Azure HDInsight Cluster hosting Apache Spark version 2.2.1. It consists of 6 NODES (2 Head + 4 Worker) with 24 cores. Head nodes are analogous to master nodes in master-slave architecture. We have two head nodes of type D12 v2, in addition to four worker nodes of type D13 v2. Each head node operates on 4 cores with 28 GB RAM and 200 GB Local SSD memory, and quantities are double those figures for worker nodes.

Dataset. For benchmarking, we choose cohorts of two datasets. The first dataset is the NY City taxicab itinerary datasets [5]. From this dataset, we choose a cohort of approximately 150k points representing a slice of data captured from taxi rides for the first half of year 2016. We have selected the green taxi trips that include, most importantly, fields containing pick-up/drop-off locational data. The second dataset that we have selected represents a cohort of 150k spatial data points that were collected through the ParticipAct project [77]. ParticipAct is a project initiated at University of Bologna in Italy, aiming at achieving the People as a Service (PaaS) vision, where people act as collectors of data that can be exploited and applied to interesting scenarios such as DBSCAN clustering. Every spatial point has a user locational data (in two-dimensional planar geometrical representations, longitude/latitude) in addition to timestamps that inform about the times of data collection.

We have applied the following intermixed parameter settings, aiming at testing the capabilities of SpatialBPE in achieving a wide variety of quality guarantees.

[5] https://www1.nyc.gov

1) ***Configurations*#1.** *Varying the DBSCAN-MR parameter settings*. We specifically vary *epsilon* and *minPoints*. We have applied two settings. 0.09 *epsilon*, 200 *minPoints*, and the other combination is 0.15 *epsilon* and 300 *minPoints*. By this test case, we aim at comparing between SpatialBPE and a baseline in their abilities to balance the tradeoffs of spatial partitioning goals, thus ultimately better trading off time-based QoS goals (e.g., latency/throughput) and other QoS goals (e.g., accuracy and resource utilization).
2) ***Configurations*#2.** *Fixing DBSCAN-MR parameter settings and varying the geohash precision.* We aim at showing the effect of adaptation (self-adaptation module of SpatialBPE) on our retrofitted version of DBSCAN-MR and its ability in achieving better QoS goals. For example, lowering the latency and maximizing the resource utilization.

4.7.5.4 Results and Discussion

All results reported in this section are the averages calculated from five query runs. DBSCAN-MR, which encloses both proximity queries that require calculating distances between points at each partition by applying a local version of a plain DBSCAN. It also encompasses a spatial join queries for joining micro-clusters resulting from local versions into global macro-clusters result set. In this thesis, we have compared the time-based QoS performance (i.e., latency) of applying our newly emerged retrofitted DBSCAN-MR version that incorporates SCAP partitioning scheme against our DBSCAN-MR version that exploits a spatial partitioning scheme from our previous work, SASAP *[74, 75]*. Our previous retrofitted version of DBSCAN-MR (with SASAP injected within its layers) is built on top of GeoSpark, whereas the current version (that is incorporating SCAP) is built atop Spark's Magellan.

We have tested SCAP against SASAP by using **Configurations#1** for five sessions each. As shown in figure 4.4 and figure 4.5, respectively, our retrofitted version of DBSCAN-MR over SCAP is adept in terms of meeting QoS time-based goals better than the previous version that exploits SASAP counterpart. Notice how an increased number of bordering replicated points (i.e., BSOs) implies a near-linear similar-pattern increase in running times of both implementations. This applies also for the case of value of *epsilon* that is equal to

0.09 and *minPoints* equals to 200. However, in both cases it negatively affects the running time of our SCAP version to a lesser extent as opposed to SASAP counterpart.

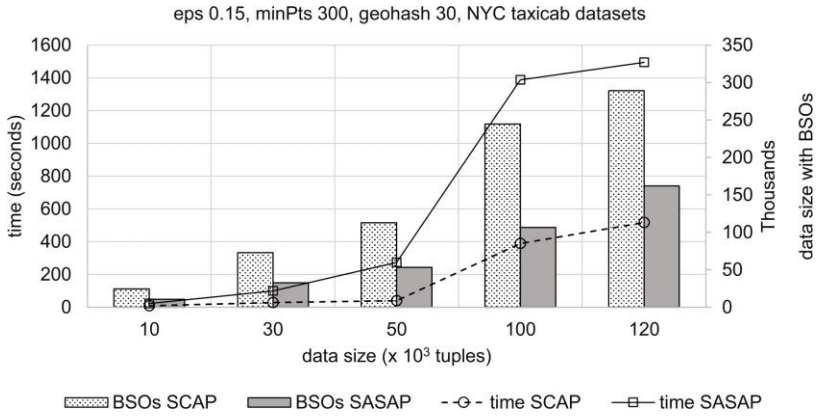

Figure 4.4. Running times and number of BSOs of our retrofitted version of DBSCAN-MR over SCAP against SASAP-based version using *epsilon* 0.15 and *minPoints* 300, secondary access on the right-hand side of the figure represents the data size with BSOs

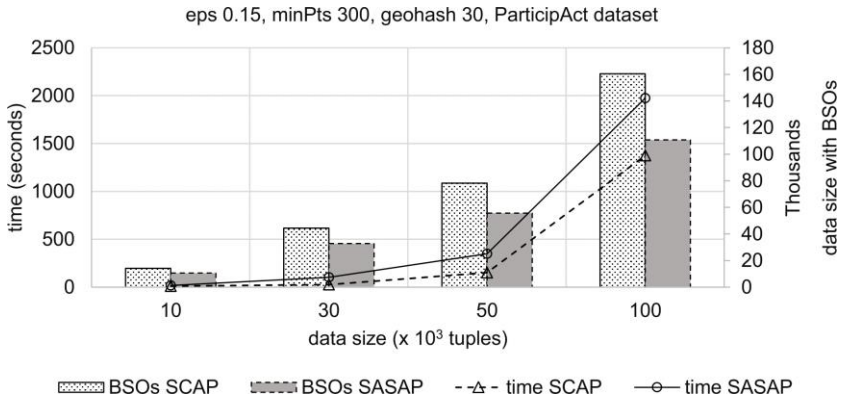

Figure 4.5. Running times and number of BSOs of our retrofitted version of DBSCAN-MR over SCAP against SASAP-based version using *epsilon* 0.15 and *minPoints* 300, secondary access on the right-hand side of the figure represents the data size with BSOs

As shown in figures 4.4 and 4.5, respectively, our retrofitted version of DBSCAN-MR can adeptly lower latency for both datasets compared to the SASAP baseline counterpart. Notice however that both versions are expensive and can reach the orders of tens of minutes for only hundreds of thousands of orders of input rate. This implies that applying DBSCAN and any variation of density-base clustering online is a façade. However, instead, DBSCAN and its variants (such as our version) can be applied in an offline stage, which is the second stage of online stream clustering (i.e., to form the final macro-clusters) [76] .

Another tweakable parameter in our SCAP method is the geohash size (i.e., precision), which determines the number of BSOs to duplicate to overlapping cells. In the second testing case, we have, on the same data cohort, applied **configurations#2**, fixing the parameters of DBSCAN-MR and enabling the controller of SCAP to tweak geohash from 30 to 35. Total running time has been slightly decreased. Fig. 4.6 depicts that tweaking geohash precision from 30 to 35 leads to less BSOs for SCAP. This is due to the fact that more geohash precision implies smaller cell sizes for cells represented by those geohashes, leading to less overlapping areas among neighboring cells. Since we depend on simply replicating objects in the overlapping zones, this would result in attenuating the BSOs count.

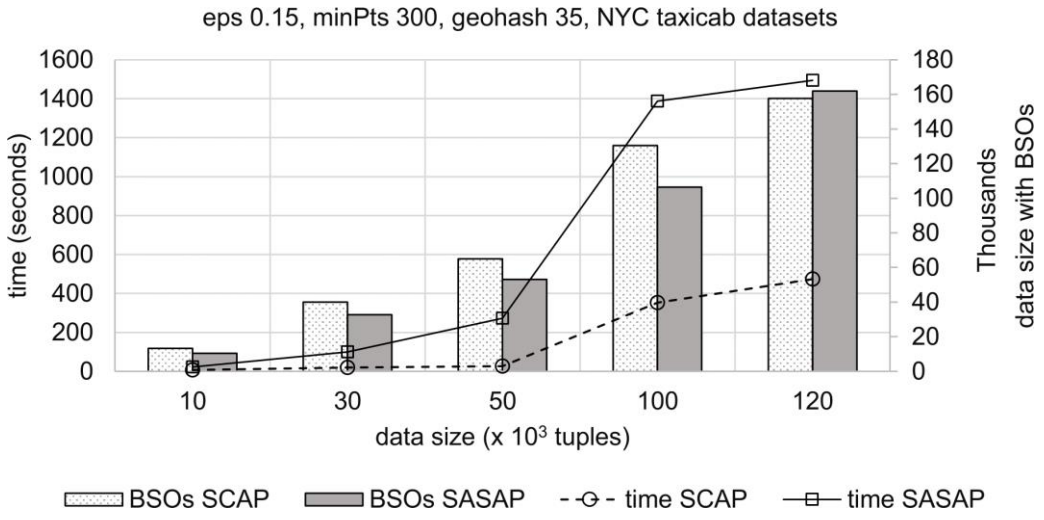

Figure 4.6. The effect of tweaking geohash precision on the number of BSOs generated by SCAP on NYC taxicab dataset. secondary access on the right-hand side of the figure represents the data size with BSOs

Notice how the running time for both competitors (our SCAP-based version and SASAP-based version) increase, but however the proportions are not the same , as the trend of SASAP-based version is to increase significantly as the number of BSOs increase, whereas the running time of our version (SCAP-based) creeps up in a smoother way. This is due to the effect of SDL preservation that SCAP is adept at achieving better than SASAP, which leads to less data shuffling. As a confirming quantification showing the benefits that we reap by applying our new method SCAP against the traditional SASAP and the associated DBSCAN-MR version, we define a speed up gain obtained through parallelization by adapting a simplified version of the Amdahl's Law [78]. More computationally, we define (4.1)

$$speedup = T_{SASAP} / T_{SCAP} \qquad (4.1)$$

where T_{SASAP} is the running time obtained by applying the baseline method SASAP to the associated retrofitted DBSCAN-MR, whereas T_{SCAP} is the running time by applying our new version SCAP to run the newly retrofitted Magellan-based DBSCAN-MR version. Our results exhibit that we gain a higher speed up through applying DBSCAN-MR with SCAP version as opposed to a lower speedup while applying DBSCAN-MR with SASAP. Figure 4.7 shows an example on the NYC taxicabs dataset. It worth noticing that the shape (bell-curve plateau-alike) obeys an important corollary of Amdahl's law. As shown in figure 4.7, there is a limit on the speedup gain (i.e., the escalation trend) that can be obtained, after which the speed up gain decreases. This is due in part to the fact that the partitioning process is sequential (i.e., non-distributed), which may become exhaustive as data size soar, putting speed up gain on a decline, even though we may still reap a discernible speed up gain. Even a low speed-up gain is handsomely beneficial in resource-constrained settings that do not offer too much of scaling-up options. Nevertheless, a future promising frontier we recommend is enabling the parallelization of the partitioning scheme, aiming at an extra improvement in the speed up gain. However, this falls outside the scope of this thesis. For more interesting results, specifically comparing our traditional method SASAP with a MongoDB version of DBSCAN-MR, readers are referred to our papers [74, 75].

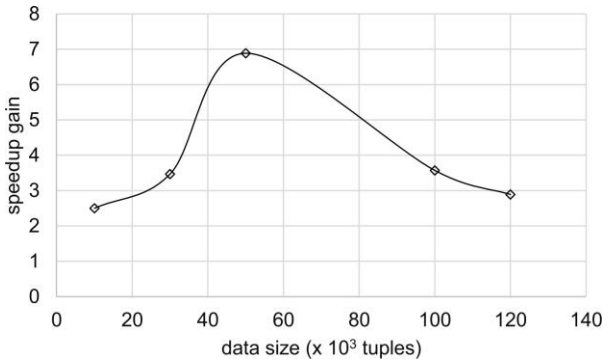

Figure 4.7. speedup by applying SCAP instead of SASAP, NYC dataset

We have decided also to quantify the gain of the adaptation of the geohash size (i.e., precision). More mathematically, we define (4.2)

$$adapGain = (BSO_{30} - {BSO_{35}}/{BSO_{30}}) * 100\ \% \qquad (4.2)$$

Where BSO_{30} is the number of BSOs generated by applying a geohash size that is equal to 30, whereas BSO_{35} is the number of BSOs by applying a geohash size that is equal to 35. Figure 4.8 illustrates how we gain roughly 44% by tweaking the geohash precision. We also dub this gain as the design effect because it results from applying the design of our SCAP scheme. Results combined show the excellence of SCAP in striking a credible balance among the three spatial partitioning goals, thus balancing loads, while also preserving spatial co-locality and minimizing BSOs.

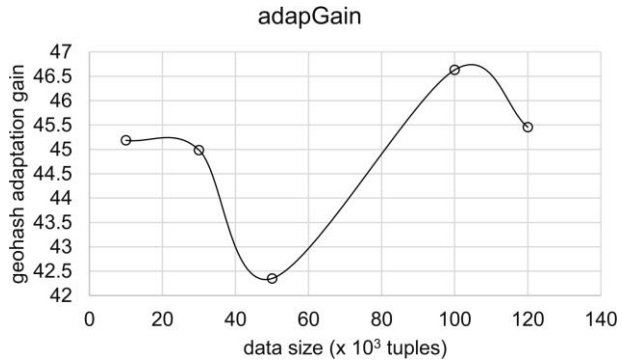

Figure 4.8. adaptation gain by tweaking the geohash precision in SCAP from 30 to 35 applied on NYC taxicabs datasets

The fluctuating shape of the adaptation gain suggests that the data skewness oscillate between different data sizes, leading at times to less adaptation gains in the middle of the shape such as the case of data size that is equal to 50k tuples as shown in Figure 4.8.

This shows the ability of SCAP controller in tweaking the geohash precision so that the resulting partitions are containing objects that are not only preserving load balance, but also preserving co-locality and minimizing BSOs. The reason of this effect is that unduly partitioned space results in partitions that cause some nodes to become congested and stragglers (i.e., nodes that take more processing times as opposed to other nodes) , and it is well known that the running time of parallel distributed systems is determined by the stragglers. Refer to our papers for more explanations [74, 75].

Figures drawn in this section support the fact that employing a custom spatial-aware partitioning with a relevant selection of configurations typically yields substantial improvements over plain spatial partitioning methods.

It worth noticing that a tension among QoS goals always show up and there is no such thing like a “free optimization that does not affect contradicting factors”. As an example, an expensive model-based approach for calculating the size of BSOs list that are candidates for replication may yield a smaller number of BSOs, thereby leading to a maximal resource utilization. This is what was achieved by our traditional method SASAP. As contradictory as it can look, a mathematically simple method may yield larger size of BSOs to replicate, thereby achieving a lower latency level at the price of lower resource utilization. That is the case of our novel method that we introduce in this thesis, SCAP. SCAP may introduce higher number of BSOs (which is a tweakable parameter that relies on the geohash precision) but it acts favorably for resource-permissive settings as it achieves lower latency than SASAP. However, for both methods, the number of BSOs to replicate relies on many factors. Most importantly, data skewness for both approaches and geohash precision for SCAP.

4.7.6 Related Works

Most works of the related art are based on Hadoop. For example, [79] propose a custom density- and spatial-aware partitioning method for pathology imaging on top of Hadoop. Their method is preserving SDL and balancing loads by relying on a costly computational

model-based approach which aims at minimizing the differences of running times between all participating partitions. On the downside, they did not account for cases where BSOs need to be taken care of. Also, Hadoop-based systems are proved to be slower than Spark-based counterparts and are currently being replaced.

On the other side, based on fast-memory structures such as Spark. Few systems have designed custom spatial aware partitioning schemes. As a case of example, LocationSpark [71] has been designed to transparently incorporate a feedback loop-based adaptive spatial partitioning scheme over Spark. Their method focuses on balancing loads by relying on an underlying model-based computational model that subsequently and periodically collects active statistics session-after-session and accordingly enhances the splitting stripes (i.e., coordinates) until the corresponding partitions that constitute stragglers vanish. The method keeps a knowledge base regarding real geometrical objects for SDL preservation. It mainly achieves this by forwarding geometrically-nearby objects to same/nearby partitions. Also, the method applies a replicate-and-refinement approach for trading-off BSO minimization.

As a recap, Hadoop-based systems are slower than Spark-based contemporaries. Also, Spark-based methods do not retrofit or integrate into density-based clustering algorithms, which then makes SpatialBPE a significant unique contribution.

4.8 SpatialNoSQL: A Scalable Storage for Spatial Data

Some workloads in highly dynamic application scenarios require storing snapshots (or even a full crawling output if resource capacity is permissive) for future offline analytics. Since streaming data sources are normally heterogeneous, it is then better to consolidate all coming formats in a unified shape, and here where NoSQL distributed storage systems come into play. They normally host data in simple JSON-like formats, treating referential integrity in simpler ways (such as embedding documents in MongoDB, refer to section 2.2.1 for details). Those systems however do not readily support sharding (i.e., splitting) natively on spatial data loads. To close this void, we have designed a scalable NoSQL-based storage system that can perform spatial-aware partitioning, and thereafter serving the spatial queries in a QoS-

aware fashion. We dub our version as SpatialNoSQL [6]. In this context, the story is a little bit different as we are accessing disk-resident data without depending on the fast memory. This is specifically beneficial in cases where snapshots do not fit comfortably in the fast memory or in other cases where a future reference for huge amount of data is required. However, the main purpose of those systems is storage, but they support query processing (analogous to RDBMSs). All in all, we should store data in a way that guarantees quality aware access, because those systems work synergistically hand-in-hand with the batch processing (SpatialBPE section 4.7) and speeding systems (SpatialSPE and SpatialSSJP in chapters 5 and 6, respectively) as a united system for interactive analytics. In other terms, they perform the heavy task of offline processing, in cases where in-memory capacity is not enough, and their slow results, also slowly changing, are served on-demand to assist in better decisions together with interactive queries. Thus, complementing the effect of QoS-aware spatial mixed workloads handling as a main contribution of this thesis.

4.8.1 Motivation

There are many dynamic scenarios in smart cities and industry 4.0 that require storing snapshots of the streaming data periodically sometimes constituting huge amounts, which calls for a scalable distributed storage system that unifies diverse heterogenous source data under one umbrella. The envisioned Industry 4.0 vision heavily depends on Data as a Service (DaaS [80]) paradigm, where avalanches of geo-referenced data loads need to be stored efficiently and quickly [81] . Scalable NoSQL ecosystems have been focusing thus far on load balancing because they know that sending geometrically co-located objects to same shards normally leads to a lopsided cluster, where heavy loads are normally clumped into few partitions. Currently, NoSQL systems (such as MongoDB) do not support partitioning on geocodes in an optimized way that can carry performance gains en-route to achieving QoS goals. In this thesis, we provide a novel viable and cost-effective spatial data partitioning

[6] The source code of SpatialNoSQL (including associated query optimizers) is available at: https://github.com/IsamAljawarneh/SpatialNoSQL

scheme for an optimized ad-hoc spatial querying in scalable NoSQL settings, forming together our SpatialNoSQL system, which is recapitulated in the next subsection.

4.8.2 SpatialNoSQL overview

Technically speaking, we have incorporated few submodules within various layers of MongoDB codebase as shown in figure 4.9.

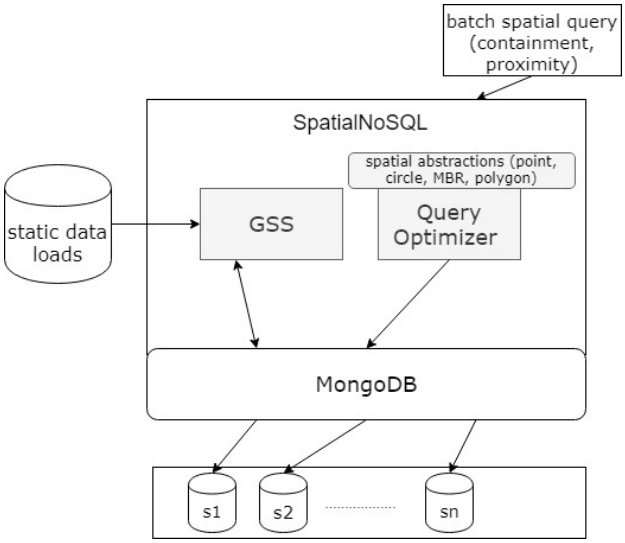

Figure 4.9. SpatialNoSQL workflow

SpatialNoSQL presents itself as a transparent layer setting between the MongoDB core and the presentation layer. It is comprised of two components:

1) **GSS** (abbreviation for geospatial sharding scheme). A custom method we design that is responsible for sharding geo-referenced data loads with the aim of striking a balance between the partitioning goals. In this thesis, we basically focus on load balancing and SDL preservation, while to a lesser extent on BSOs minimization. By this we aim at a significant effect that strikes a discernible balance between the QoS goals. The peculiarities of GSS are illustrated in section 4.8.3.
2) Retrofitted **query optimizers**. Our version takes full advantage of our partitioning method GSS in supporting spatial queries that intrinsically incorporate spatial joins, such as proximity-alike and containment queries with quality guarantees as explained in section 4.8.3.

4.8.3 QoS Aware Spatial Data partitioning for NoSQL

In simpler terms, storage-oriented data partitioning means disseminating (a.k.a. sharding) datasets to multiple nodes in a distributed storage environment [82]. We specifically focus on striking a balance between load balancing and SDL preservation. This complies with the types of the spatial queries that we support for NoSQL as explained shortly in section 4.8.4. Natively, MongoDB supports quadtrees and z-curves indexing. However, as per 4.0 version, those are utilized for indexing only and not sharding.

To strike a balance between the contradicting QoS goals (such as low-latency and high accuracy), we have designed a novel spatial sharding scheme for MongoDB, which we dub as Geospatial Sharding Scheme (GSS for most of the rest of discussion). GSS aims at preserving spatial characteristics and load balancing. As the time of this work, MongoDB does not offer native support for geocode-based spatial data sharding. Figure 4.10 elucidates the workflow of GSS, which is formally expressed in Algorithm 4.2. The algorithm proceeds as follows. It first accepts geo-referenced documents (representing spatial objects) as an input and thereafter employs a simple mapper on them to incorporate a geohash field. Afterwards, documents are clumped into small chunks by relying on their associated geohash values. As of yet particularly happens the load balancing, where overburdened chunks are split. GSS then advances by employing a loader that sends chunks to their relevant shards. By doing that, SDL preservation is guaranteed to a good degree, and also load balancing is traded off appropriately altogether. In addition, this method assures a minimal shuffling (*chunk migration* in MongoDB parlance) during query running.

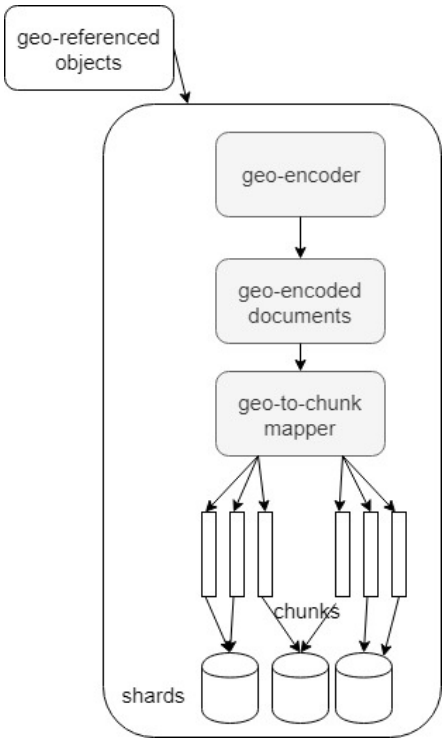

Figure 4.10. GSS sharding scheme

Algorithm 4.2 GSS sharding scheme for NoSQL frameworks

***/* input:** two-dimensional spatial points on the form of (longitude, latitude) received from GPS-enabled devices */*

1: **Foreach** point p in points
2: geoCode ← geohashEncode(p) *//geo-encode a spatial point using geohash*
3: shardID ← geoMapper (geoCode) */* assign a shardID to which spatial point should be sent */*
4: chunk [shardID]. add (geoCode) *//add geocoded spatial point to the appropriate chunk*
5: load_chunks(shards[1…i]) *//bulk loading chunks altogether to their relative shards*
End foreach

4.8.4 Spatial Query Optimizers for NoSQL Scalable Distributed Storage

4.8.4.1 Spatial Query Primitives Supported

We support two primitive types of geospatial queries:

1) **Proximity** queries. For example, spatial range search and *k*NN.
2) **Containment** (a.k.a. inclusion). We support two kinds of containment searches:
    - Containment searches based on arbitrarily-shaped embedding areas (i.e., polygons). Those need Point in Polygon (PIP) tests. We refer to this type as containment-PIP to distinguish it from the other types.
    - Containment searches based on *regularly*-shaped embedding areas (i.e., circles), we refer to this category as Point-In-Circle (PIC for short) test, which is analogous to PIP test with the exception that the embedding area we are searching in is circular, thus retrieving *concentrically* located points.

4.8.4.2 NoSQL Query Optimizer Overview

Traditional spatial query processors obey the scatter-gather scanning scheme by performing exhaustive searches. However, few systems such as MongoDB encapsulate routers that forward the query request to specific shards based on the sharding key. However, Spatial indexing is not natively offered for sharded collections in MongoDB [7].

Aiming at closing this void, we have designed a NoSQL query optimizer for MongoDB, as depicted in figure 4.11 (shaded components represent our patches), specifically for assisting spatial proximity (such as *k*NN) and containment queries in exploiting the merits of GSS sharding scheme, thus achieving a prespecified set of QoS goals. By doing so, we focus on higher resource utilization, higher throughput, lower latency while keeping the accuracy untouched. Chiefly, we have retrofitted a version of the plain MongoDB spatial join query optimizer, which is used specifically for queries that incorporate containment, intersection or overlap spatial predicates. Our retrofitted edition exploits our newly introduced

[7] https://docs.mongodb.com/manual/core/2dsphere/

partitioning scheme GSS for supporting queries that intrinsically encapsulate spatial join predicates such as ensembles (e.g., Top-N) and inclusion.

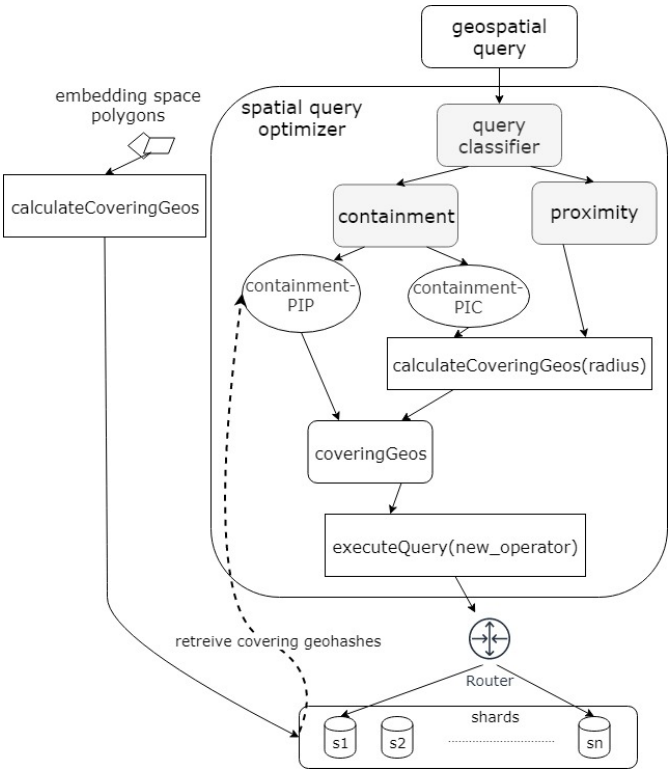

Figure 4.11. Spatial-Aware Query Optimizer for NoSQL

The query optimizer starts by the same procedure for both query types (i.e., proximity and containment). The optimizer first classifies the query to decide based on the type either to retrieve the stored covering geohashes (in case of containment-PIP) or to calculate the geohashes based on the embedding circle (in case of containment-PIC and proximity). The procedures are different in both cases.

For containment-PIC and proximity, we utilize our legacy support that appeared in our paper in [4]. The procedure first constructs a circle (given the radius and a query point), then a MBR for the circle is imposed, and thereafter a list of covering geohashes is generated based on the MBR. On the other hand, for containment-PIP, we utilize our new support [83] depending on a precalculated geohash covering for the embedding space, where we have a list of polygons (neighborhoods, boroughs or districts in city management terms) and we

calculate the covering for each polygon, then we store the result coverings in disk. This is a one-time process that is cheaper than calculating coverings for every query independently. Our optimizer then reformats the query operator so that it encapsulates the geohashes covering as a prefiltering stage (a specifier in MongoDB parlance), which then acts as a pruning machine that significantly reduces the search space. Thereafter, the new formatted query is passed to MongoDB query router, which then forwards requests to only shards that contain the candidate results.

In our legacy work [4] , we have provided two new operators for proximity-alike and containment-PIC queries over MongoDB. In short, we have provided a support for proximity-alike queries via a retrofitted version encapsulated within the plain MongoDB layers, which then executes as a MapReduce job. At the time, $near or $nearSphere MongoDB plain operators were not operating on sharded collections, a drawback that prohibits them from exploiting the benefits of distributed processing. However, starting from MongoDB 4.0, $nearSphere operator has started operating on sharded collections. Consequently, to further extend our legacy support, we have decided to extend the support for proximity-alike queries by employing geohash coverings as a specifier on a $nearSphere operator this time. In this thesis, we show our latest support for the proximity-alike queries [83] . Interested readers are referred to our paper [4] for more information about our legacy method for supporting the proximity-alike queries and the related interesting results.

Containment and proximity queries normally exploit geospatial indexes such as MongoDB 2dsphere. As such, spatial join is pivotal so that a list of spatial objects that are encompassed within the fences of a geometrical covering is generated. Algorithm 4.3 summarizes our Spatial join optimizer for NoSQL workflow. We have introduced a novel geohash specifier that works as a quick-and-dirty sieve (i.e., a prefiltering stage). This acts as a pruning device that prunes aggressively the search space prior to applying an expensive PIP test.

Algorithm 4.3 Spatial join optimizer for NoSQL workflow

1: **Input:** two versions:
Either `Query: q, points: p, r: radius, qp (longitude, latitude): query point` for `proximity through $nearSphere operator`
OR `Query: q, points: p, neighborhoods: nb for containment-PIP through $geoWithin with a geometry specifier`

2: coverGeo ← getCoverGeo (embedding_area, geoPrec) */* List of geohashes covering region (circle or polygon) embedding_area is either polygonal arbitrarily shaped neighbourhoods (nb in the input) or a regularly-shaped circle (with radius r in input) */*

3: coverGeoSpecifier = "geohash": `{"$in": [coverGeo]}`

4: newOperator = add (coverGeoSpecifier, MongoDB_operator) *//adding the geohash specifier to the plain MongoDB operator*

5: p.createIndex(`({"geohash":1,"location":"2dsphere"})/* two-levels indexing scheme */`

6: executeQuery (q, newOperator,p) *//execute the query using the new operator*

Geohash is a geospatial encoding scheme that normally generates a single-dimensional representation as a string that encompasses a geographical meaning. MongoDB recognizes geohash string as a textual field. This is a free optimization that allows using geohash encoding as a pruning machine considering the fact that geohash is also used for partitioning (i.e., sharding) where we select geohash filed as a sharding key. The resulting spatial index that is then imposed is a composite key in the sense that it is composed of geohash field that we provide and also the 2dsphere that is already provided by the plain MongoDB query optimizer. This mechanism assures that both indexing schemes synergistically reinforce each other without their limitations. To take a more serviceable perspective, a composite index that is comprising geohash and 2dsphere ensures that we enforce spatial indexes on a two-levels basis, local and global. In one hand, geohash indexing acts as a global index that is beneficial as it is the sharding key that assists the query router in pruning significantly the search space. On the other hand, 2dsphere acts as a local index that is applied for each shard independently, helping in further pruning the search space as it only examines those documents locally that in real geometries are fenced within the boundaries of S2 coverings. In MongoDB compound indexing strategy, the order of the indexes matters (those are the indexes that constitute the compound index). Having said that, because we have specified

the geohash indexing as the first index in the composite index, we could reap many benefits and enable pre-pruning the search space to highly plausible magnitudes.

Providing a more heuristic overview, our methodology acts in the following manner. First, we overlay the embedding space with a fixed-grid network. Afterwards, we enforce an ordering representation (z-curves and specifically geohash) so that we help in reducing the dimensionality of the underlying multidimensional embedding space. This process results in a z-order that helps in determining the order at which the covering grid cells are visited while answering a query request. This causes a substantial pruning for the search space and returns a list of points that interact with the covering. The result is then forwarded to the second part of the compound index, 2dspahere that is freely provided by MongoDB, which additionally prunes the local spaces. 2dspehere first linearizes the embedding space (which is a portion of the pruned space that resulted by using the global index geohash). Afterwards, MongoDB imposes an access structure (specifically, a B-tree index) on the sub-coverings. Running times involved in the two parts of our procedure are linear and independent from the total size of a MongoDB collection. This preprocessing mechanism causes MongoDB compiler to read points (i.e., documents in MongoDB terms) that only interact with the geometrical coverings, thus significantly minimizing the unnecessary overhead that may be caused by costly frequent I/O operations.

4.8.5 Experimental Setup and Parameter Settings

Environment. We run SpatialNoSQL on a MongoDB Atlas cluster deployed on Microsoft Azure cloud hosting a newer version of MongoDB (specifically version 4.0). Our deployment consists of 4 shards. Each shard has the following profile: M30 tier with 32 GB storage, 8 GB RAM and 2 vCPUs.

Datasets. For benchmarking, we choose to use the NY City taxicab trips datasets [8]. We have selected a cohort of two months dataset (that is constituting around three million units), representing data captured through taxi itineraries for the first half of year 2016. We have

[8] https://www1.nyc.gov

selected the green taxicab trip records, which include interesting fields capturing, most importantly, pick-up/drop-off locations and trip distances.

Parameter settings.

- *Varying geohashes precision* and comparing total documents and keys examined in addition to the time-based QoS goals such as the running time. We have applied this setting to compare the application of our optimization for the containment-PIP (i.e., based on the PIP with polygon geometry specifier) test against the plain MongoDB support. In addition, we have applied this setting for the Top-N queries (those that are a special case of containment-PIP).
- *Varying the circle radius* and measuring total keys and document examined in addition the time-based QoS goals such as the running time. We have applied this setting to compare the application of our optimization for the proximity queries based on $nearSphere MongoDB operator with a test point and circle geometry specifier.

4.8.6 Test Cases, Results and Discussion

All results reported in this section are the averages of running same queries with same settings for five times.

4.8.6.1 Testing Containment-PIP Query Optimizer

In this thesis, we focus on containment queries that demand a PIP test.

Query. We have tested based on the spatial containment-PIP query: “find all taxi trips that have been originated in a given neighborhood in NY City during a two months period”. Figure 4.12 depicts that our newly introduced optimizer significantly outperforms the vanilla MongoDB optimizer for containment-PIP test.

Notice that for a geohash precision that is equal to 35, our geohash-based query optimizer searches three shards only (instead of the scanning the four deployed shards). On the contrary, the plain MongoDB optimizer requires scanning all the four shards of the deployment, which causes an extra overhead. It is also apparent that the number of documents and keys that need to be examined by using our optimizer are less than those that need to be examined through the plain MongoDB version. This fact applies to most geohash values. However, for narrower geohash values such as the case of geohash value that is equal

to 25, both methods are on the brink of the need to examine the same number of documents and keys. This apparent paradox is in part due to the fact that a smaller geohash precision implies necessarily a wider geohash coverage (embedding space that is covered by a specific

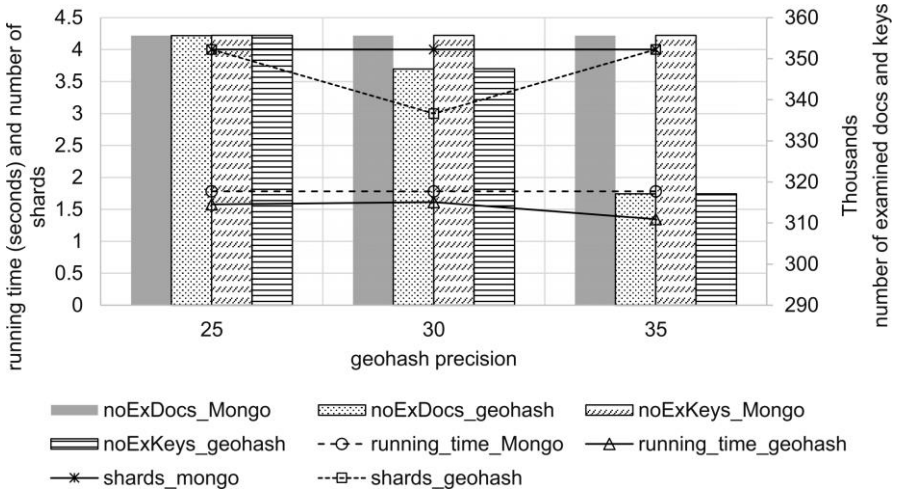

Figure 4.12. Comparing the performance of our new spatial join query optimizer on containment-PIP queries (with a $geoWithin operator with a geometry specifier) against the vanilla MongoDB optimizer. 'Mongo' in the legend means the plain MongoDB, whereas 'geohash' means our new geohash-based optimizer. noExDocs and noExKeys mean the number of examined documents and keys, respectively

geohash precision expand as we narrow the geohash value). For example, a geohash precision 25 covers a cell that is less than 4.89 kilometers and almost 4.89 kilometers in width and height, respectively. On the other side, a bigger geohash value such as a value that is equal to 30 covers 1.22 kilometers by 0.61 kilometers for width and height, respectively, which is a smaller area. In simpler terms, larger geohash value implies a smaller area, hence the overlapping space (between many geohashes) shrinks, which means that less documents (i.e., spatial points in real geometries) fall within the fences of those corresponding geohash coverings. This fact is proved also with the case of geohash precision that is equal to 35 as shown in figure 4.12. As a way of contrast, smaller geohash values mean larger areas, and thereby more documents fenced within their boundaries, which causes more documents to be examined at run time.

To quantify results in a more coherent way, we have calculated the speed up by relying on Amdahl's law as in (4.3).

$$speedup = {T_{mongo}}/{T_{geohash}} \qquad (4.3)$$

Where T_{mongo} is the running time by applying MongoDB plain operator, whereas $T_{geohash}$ is the running time by applying our version based on geohash.

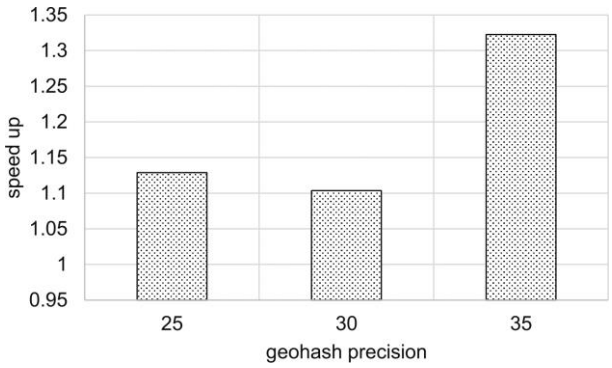

Figure 4.13. The speed up gain we obtain by applying geohash-based containment-PIP optimizer against MongoDB plain optimizer

Our results shown in figure 4.13 prove that we always (for all geohash precisions) gain speed up by applying our filter. Notice that we obtain a better speed up by conveniently tweaking the geohash value. For example, we obtain the best speed up by using a geohash value that equals 35 which further strengthens the argument we started beforehand regarding the results of figure 4.12. However, all in all, our optimizer outperforms the plain MongoDB counterpart by numerically significant orders.

4.8.6.2 Testing Top-N Query Optimizer

By this test scenario, we compare the ability of our optimizer in striking a better balance between the QoS goals.

Ensemble queries such as top-N are special cases of containment-PIP. Having that in mind, we can answer Top-N queries by checking the spatial objects that are fenced within the boundaries of each neighborhood (i.e., polygon). This in its essence requires applying the containment-PIP test operator for each object that interacts with the coverings.

Query: We have tested our containment-PIP optimizer effect on top-N queries based on the following query: “what are the top-10 neighborhoods in NY City, USA that have the most taxi pickup orders in a period of two months”.

Figure 4.14 depicts that the vanilla MongoDB optimizer underperforms our novel optimizer for all geohash values. As it is obvious, the best case happens at geohash precision value that is equal to 35, which directly implies that geohash precision is a pivotal tweakable configuration parameter in our optimizer.

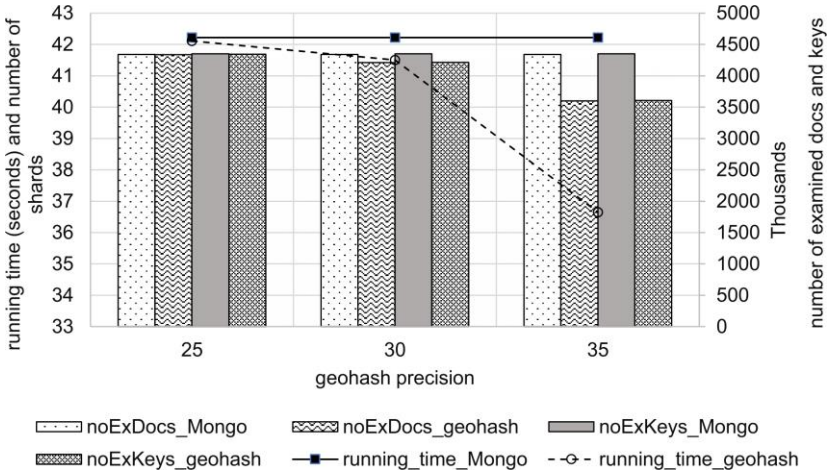

Figure 4.14. Comparing the effect on performance of our new containment-PIP query optimizer on ensembles (specifically Top-N queries) against the plain MongoDB optimizer. Mongo in the legend means the plain MongoDB, whereas geohash means our new geohash-based optimizer. noExDocs and noExKeys mean the number of examined documents and keys, respectively

Expressing those results another way, we apply a simple speedup formula as in (4.4).

$$T_{topN\_mongo} \Big/ T_{topN\_geohash} \qquad (4.4)$$

Where T_{topN_mongo} is the running time by applying the plain MongoDB containment-PIP optimizer on Top-N queries, whereas $T_{topN\_geohash}$ is the running time by applying our geohash-based containment-PIP optimizer on Top-N queries. Figure 4.15 depicts what has been sketched. Those results show that our query optimizers are able to satisfy and trades off time-based QoS goals (specifically running and latency times in this case) better than the plain versions. The tiny-gain case where only tiny speedup is obtained when applying the

geohash precision value that is equal to 25 complies with our argument in section 4.8.6.1 regarding figures 4.12 and 4.13.

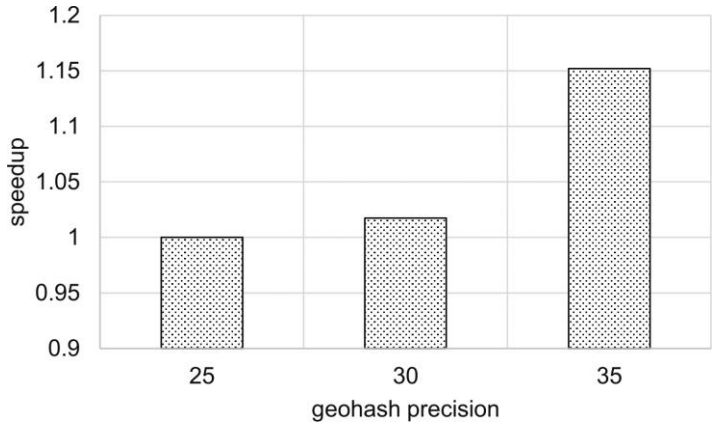

Figure 4.15. speed up by applying geohash-based containment-PIP optimizer against MongoDB plain optimizer

To quantify deeper, we apply (4.5) to calculate the resource utilization gain (i.e., CPU running times)

$$deff = gain = (SO_{\text{Mongo}} - \text{SO}_{\text{geohash}})/SO_{\text{Mongo}} \qquad (4.5)$$

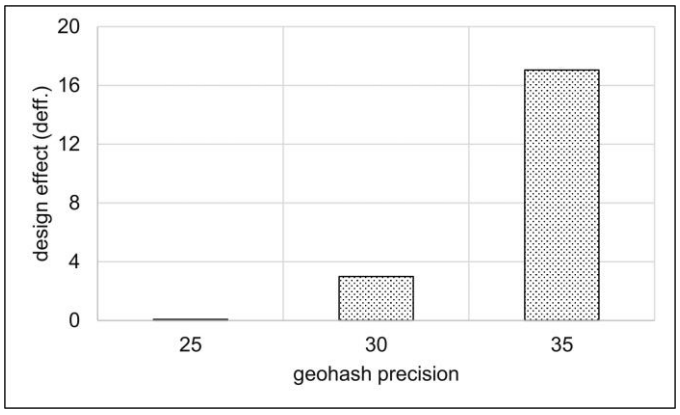

Figure 4.16. Design effect expressed as a resource utilization gain

, where $deff$ is the design effect (i.e., gain), SO_{Mongo} is the number of scanned documents by applying the plain MongoDB optimizer, whereas $\text{SO}_{\text{geohash}}$ is the number of scanned units by applying our version (geohash-based containment-PIP optimizer). Figure 4.16 shows the gain we obtain. This specifically achieves the higher-resource-utilization QoS goal, as the number of CPU cycles reduces significantly by avoid scanning unnecessarily objects that are

not contributing toward a result set. Notice again in this case the fact that we obtain higher design effect (plausible) by applying a geohash that is 35 as opposed to narrower values such as those that are equal to 25 and 30.

*4.8.6.3 Testing Proximity Queries (for example, k*NN*) Optimizer (relying on a retrofitted $nearSphere MongoDB operator with a test point and circle geometry specifier).*

Query: We have tested our proximity query optimizer effect based on the following *k*NN query: "find all locations of taxi itinerary pickup orders within a predefined circular distance from a concentrically located test point within a period of two months, sorted from nearest to farthest".

As shown in figure 4.17, Our novel method outperforms that of MongoDB plain by discernable margins. However, we have noticed that in cases that require examining a substantial number of documents and keys, the running times reduction gain may vanish. For example, notice the case where a prespecified distance is a radius that is equal to 15 kilometers. In that case, our proximity query optimizer requires examining a number of documents and keys that is roughly similar to those required by the plain MongoDB. This however is normal and healthy because the number of returned spatial objects that satisfy the distance predicate (i.e., 15 kilometers) roughly equals the total number of documents in the original points collection.

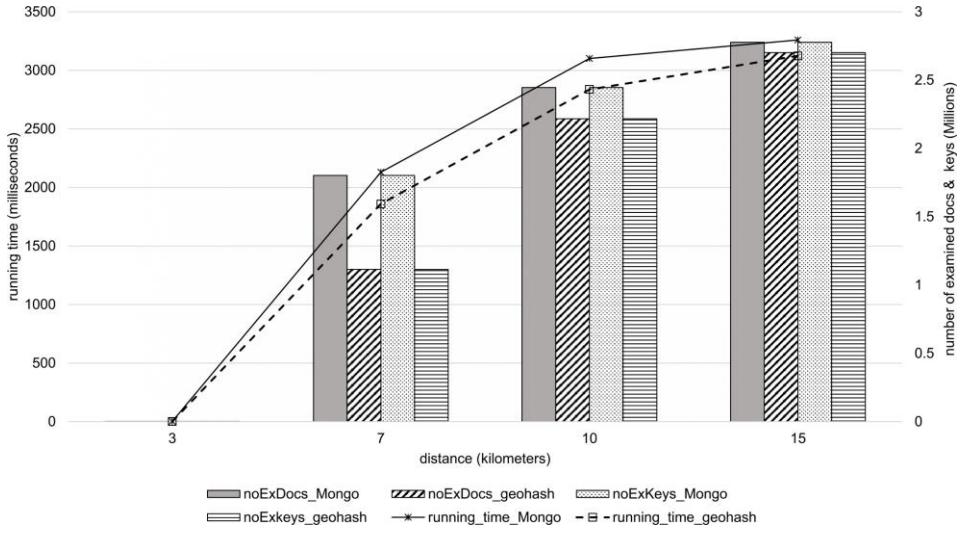

Figure 4.17. the performance of our spatial join query optimizer on proximity queries (with a $nearSphere operator) against the plain MongoDB optimizer. Mongo in the legend means the plain MongoDB, whereas geohash means our new geohash-based optimizer. noExDocs and noExKeys mean the number of examined documents and keys respectively

Results appear in this thesis include our novel supports [83] . For our legacy supports, including containment-PIC optimization, in addition to the legacy MapReduce-based support and results of the proximity query optimizer, readers are referred to our paper [4].

All results shown in this section prove that SpatialNoSQL is adept in satisfying Qo[4]S goals. Specifically, we have focused on time-based goals such as low-latency, in addition to other goals such as higher resource utilization and high accuracy. It does so by applying GSS with retrofitted query optimizers for both proximity (such as *k*NN) and containment queries. GSS achieves a significant weighted balance between two partitioning goals, SDL preservation and load balancing. GSS does not consider BSO minimization as proximity and containment queries does not require the replication of BSOs to neighboring grid cells, thus not inducing any overhead.

To quantify at a cursory level, we apply (4.5) to calculate the resource utilization gain (i.e., CPU running times) that we may reap by applying our proximity optimizer instead of the default MongoDB counterpart.

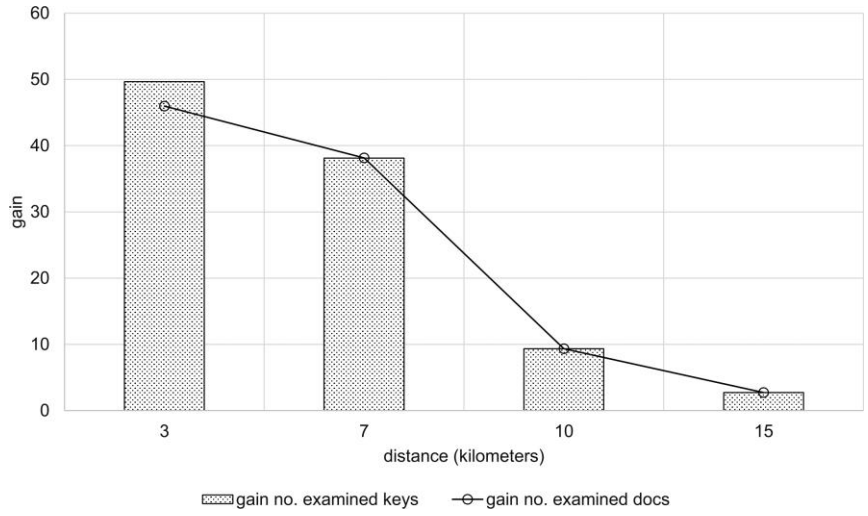

Figure 4.18. Design effect expressed as a resource utilization gain

As shown in figure 4.18, the gain we can obtain inclines linearly as we increase the radius of the area. This complies with our discussion regarding the results of figure 4.17. However, we always obtain a gain by applying our optimizer against the plain counterpart.

4.8.7 Related Literature

From the relevant literature, we herein list some few works. For the column-oriented databases, [84] have designed a sharding scheme they term as SPPS, which basically aims at balancing loads while preserving SDL. They employ a model-based formulation for computing the number of relevant partitions that are required for load balancing. In addition, they employ an indexing scheme known as spatial longest common prefix (SLCP for short) for geo-encoding spatial objects in a manner that achieves SDL preservation, and thereby sending real-geometrically nearby spatial objects to the same partitions.

In the same vein, [85] have designed a framework they term as HGrid on top of the HBase database system. HGrid works by a mixture of a quadtree and grid-based representations, aiming at basically achieving SDL preservation. Also, [86] have designed a scheme that is based on a method known as GeoSOT [87] over HBase. GeoSOT partitioning method is similar to a multi-level geo-encoding scheme that incorporates a micro level (on the scale of square centimeters) to bigger macro levels. Grid cells are overlay on each level and a z-curves ordering is further imposed on the grid to hasten the order of access, thus accomplishing a credible balance between load balancing and SDL preservation. Range spatial searches are supported. Perhaps most importantly is a work known as GeoSharding [88] , which encompasses a method that transforms the embedding space into a virtual network of shards, such that each (or few) shards correspond to an area in real geometries. Each time the system receives a spatial point, it emits it to the shard which corresponds to its real-geometrical location, thus preserving SDL to a plausible degree. They have employed Voronoi indexing because it covers irregularly-sized polygons (i.e., regions), which is in contrast to z-order curves. GeoSharding is engineered atop MongoDB. The picture thus far that has emerged from the literature is that no single splitting scheme is a panacea. Instead, several approaches should be combined and tightly coupled so that they synergistically produce a credible method that can be used for complex scenarios in dynamic applications. By designing GSS over MongoDB and all the associated optimizers, we have specifically achieved that goal.

4.9 Chapter Conclusion

A weighted balance should be considered for conflicting data partitioning goals, SDL preservation, BSOs Minimization and load balancing. Those are conflicting in a way that makes a closed-form solution NP-hard and far from being solved (rendering the problem computationally intractable at times depending on the data distribution and skewness). Exaggerating the optimization of any of those competitors, even counterintuitively with small factors, can carry over a negative effect on other goals. We recommend seeking to strike a plausible balance between partitioning goals, while combining that with custom query optimizers that exploit the novel sharding methods in a way that assists the system to achieve QoS goals. Considering also that most partitioning methods are performed as sequential jobs in distributed systems, the gains by a custom partitioning procedure should mitigate any additional overhead it induces. Consider also that as per the Amdahl's law, there is a limit on the gain (especially speedup in data parallelization scenarios) that can be obtained through an optimization for a parallelly executed job that intrinsically incorporates a non-dispensable sequential part. Amdahl's law [44] is a dominant corollary in this context, where the incremental gain obtained by continue optimizing the same portion vanishes ultimately. Having said that, the non-separable sequential data partitioning part should be minimized, and any custom partitioning method should seek trading off the three partitioning goals in a convenient manner that does not add superfluous overhead to the equation.

In accordance with those recommendations, SpatialBPE and SpatialNoSQL perform favorably against baselines. Resource utilization also should be considered a QoS goal with a high priority, thus striking a balance between time-based QoS and economy-based goals is pivotal.

In summary, we posit that spatial partitioning plays a vital role in the speed of spatial query processing in parallel computing environments. However, spatial-aware data partitioning alone is unable of achieving all desired qualities.

Our focus in this chapter was on batch processing and scalable storage and the systems we design have taken central position in the SpatialDSMS, thus easily finds their way to break into big geospatial management domain. An upcoming stage in the pipeline could be accepting data as streams, then joining those with historical archives. This enables

SpatialNoSQL and SpatialBPE to engage in mashup workloads effectively. Also, it is infeasible to store on-the-fly deluge of geo-referenced datasets, despite the need to store snapshots at times. Interactive processing is receiving a momentum for the better part of the last decade or so. Structures, models and algorithms from the batch processing space, some, are transferable while others need more efforts to be considered for online speedy processing of geo-referenced datasets. For example, some partitioning spatial methods while performing with QoS guarantees in batch mode cannot be applied to online data as they take a huge toll on I/O performance. Having said that, online processing despite complementary to the other parts in our architecture SpatialDSMS (refer to section 3.4), has its own peculiarities that should be considered which are covered throughout the next two chapters.

Chapter 5

SpatialSPE: Spatial Approximate Query Processing

Nowadays, with the abundance of cheap GPS-enabled devices, IoT is emitting avalanches of geo-referenced data streams. Most applications at the top level are seeking insights by presenting data in a multidimensional manner (dashboards, heat maps, and other visualizations) interactively so as to serve them to higher level managers for an improved decision making and strategic planning. It is then important to present those insights interactively in a timely fashion before they become obsolete and loses their value. However, the 3Vs of big data (velocity, variety and volume) challenges the capacities of current SPEs. Provisioning extra computing resources in a dynamic allocation style is often the solution that is becoming a norm in state-of-art SPEs. However, scaling that way often enforces a huge toll on the QoS goals and is not able to strike acceptable margins of balancing between time-based QoS goals (i.e., low-latency, mostly on the scale of sub-seconds) and other goals such as high resource utilization. As a way of coping with that, load shedding lends itself as a highly desirable solution, especially knowing that a well-designed load shedding mechanism yields statistically plausible results with rigorous error bounds. This is the essence that encouraged us to prefer approximations over dynamic allocation approaches (i.e., elasticity). In addition, elasticity induces extra overheads through repeated reconfigurations that may require various cycles of shutdowns and restarting which slides an effect that negatively impacts an end-to-end QoS metrics (e.g., latency).

After building SpatialBPE and SpatialNoSQL which support faster analytics in batch mode (in-memory and disk-based processing), we have realized that despite we have obtained orders of magnitude gain over state-of-art counterparts, the implementation is still suffering slowness (on the orders of minutes for complex analytic scenarios such as DBSCAN) and cannot be applied “as-is” for processing data in-flight (a.k.a. arriving in online settings). The fact that in spatial intelligence, scientists accept approximations with rigorous error bounds encouraged us then to search for gaps where we can contribute by providing spatial-aware optimizations for Approximate Query Processing (AQP) in online settings. This led to the design and implementation of a baseline engine we dub as SpatialSPE, a unique and solo in

its class that can provide an SQL-like (exposing micro-batches through a declarative API similar to SQL in relational DBMSs) interface for distributed Spatial Approximate Query Processing (SAQP). With SpatialSPE, we support an extra set of spatial analytics coming this time from the spatial statistics (a.k.a. geo-statistics) field, aiming at enriching the pipeline with a diverse set of analytics that meet the requirements envisioned throughout the motivating scenario of section 1.1. To the best of our knowledge, we are not aware of any system in the relevant literature that achieves the goals of SpatialSPE.

This chapter is organized as follows. We first motivate the work in § 5.1, this is followed by a primer on theoretical foundations in § 5.2, then we discuss the design of SpatialSPE with the associated SAOS algorithm in section § 5.3. We then in § 5.4 show the technical details behind the realization of SpatialSPE in short, followed by a discussion of the results obtained by applying SpatialSPE in § 5.5. Thereafter, in § 5.6 we recapitulate important works from the related literature, and then conclude the chapter in § 5.7 with a short forward introducing the need for a complementary work in chapter 6.

5.1 Motivation

The widespread adoption of IoT devices have caused avalanches of geo-referenced data streams to flow endlessly and feed DSMSs, and specifically SPEs [22]. The timely exploration of those streams offers deep insightful analytics that assist strategic planning in all aspects of our lives, including city planning, urban computing and health care. *Low-latency* and *high-estimation-quality* are the two greatly antithetical QoS goals that need to be trade off in a plausible way. Deterministic solutions, where exactness is required, cannot normally strike a plausible balance between those contradicting QoS goals. Thus, Approximate Query Processing (AQP) lends itself as an alternative probabilistic path that has shown promising in striking a balance between QoS goals. The fact that, more than often, users are willing to abandon tiny error-bounded estimation quality by accepting a small reduction in the gain profit margin for the benefit of even a small latency gain. In other terms, it is important to comprise an acceptable degree of exactness but on the price of avoiding the slowness induced by an exhaustive search, thus striking a balance between conflicting QoS goals (such as the case of low-latency against high-resource utilization). AQP depends on

many data size reduction techniques, from which sampling presents itself as a leading solution. Sampling means selecting a portion of the total data (i.e., population) and compute an error-bounded statistic based on that portion. A great challenge relates to designing a sampling scheme that is able to select representative samples that yield estimations with rigorous error-bounds [89]. Most online sampling methods embrace randomness, by depending on sampling schemes that are based on random sampling. However, most interesting data are highly skewed (as opposed to the normal distribution). Designs that are based on randomness proved inefficient for non-uniformly distributed data such as geospatial data. In real scenarios, data streams are geo-referenced and being attuned to this characteristic in every aspect of the DSMSs is essential for location intelligence to success, including the online sampling scheme. Aiming at closing those gaps, we have designed and implemented SpatialSPE (short for Spatial Stream Processing Engine), together with a specialized online sampling method SAOS (discussed shortly in section 5.3.4). Our contributions by introducing SpatialSPE are the following. First, we have designed a fast in-memory first-in-class online spatial sampling scheme and incorporated it with an emerging SQL-like based micro-batch SPE, Specifically Spark Structured Streaming [6], (SpSS as a shorthand). We dub our method as SAOS (explained in section 5.3.4). The originality of our method lies in the fact that it is able to pick interactively spatially proportional representative samples that, when used in an approximate yield results with high quality. The second contribution we provide through SpatialSPE is that we have retrofitted the SpSS query incrementalizer so that it becomes aware of the spatial approximate queries that are confronting the system up the pyramid. We use the retrofitted version to incrementalize geo-statistical computations on geospatial data. Incrementalization means that results accuracy will be improving stepwise. Queries include single spatial queries, such as approximating a study variable (e.g., the '*average'* or '*total'* of a variable). We also support spatial online aggregations, such as Top-N rank geo-statistics. To the best of our knowledge, we are not aware of any system from the relevant literature that achieves these goals.

5.2 Theoretical Foundations

In this section, we aim at laying down the foundation for delineating coherently the ideas presented thereafter. We discuss various sampling designs and the need for spatial-aware methods that consider spatial patchy distributions by design.

5.2.1 Stream Processing

Stream Processing can be loosely defined as any middleware that is responsible for processing streaming loads of data, aiming at gaining deep insights. Those systems normally have a topology of operators, often known as Directed Acyclic Graph (DAG), which also comprises input streams that emit data and the output sinks that receive the (often) incrementalized results. In distributed deployments, operator instances are replicated through the network so as to distribute the workload. The goal then is to achieve a prescribed list of primary QoS goals such as low latency, high throughput and high accuracy. In addition to secondary QoS goals (e.g., load balancing) that are defined for empowering the primary goals. Those goals are normally achieved through elasticity or approximate computing.

Parallelizing SPEs is important for achieving QoS goals (i.e., lowering latency and gaining throughput), where the system depends on executing multiple instances of the same operator (i.e., one in each worker node) on a subset of data in a parallel fashion (as opposed to the traditional sequential execution). Two processing models are common in SPEs, record-at-a-time (a.k.a. tuple-by-tuple) or micro-batching. In the former, as its name implies, each record is processed independently in the sequence it arrives, whereas in the latter, multiple records are accumulated into micro-batches before being sent to parallelly distributed operator instances. Stream processing has borrowed many semantics from the batch processing because of micro-batching model. It also introduced a new set of semantics that are not required in batch mode. For example, window semantics either constraining the period (i.e., time-based windows) or number of records (i.e., count-based windows) that can collectively be processed in one shot. In this thesis, we focus on tumbling windows, thus reducing the endless processing mechanism of a stream by discretizing (a.k.a. windowing) it into more manageable finite subsequence periods that are non-overlapping (i.e., tumbling). A stream tuple belongs only to one window period in tumbling semantics. Aggregations then are performed on each window independently or with a state management mechanism in case of

stateful aggregations. Thereafter, results are served interactively and incrementally (i.e., stepwise) to the user.

5.2.2 Sampling

5.2.2.1 A Short Primer on sampling

In statistics, sampling is loosely defined as the procedure of selecting a representative portion (could be miniatures) of a population for estimating an unknown population quantity, such as an '*average'* or '*count'* of a target variable. Population represents all units in a specific study area. For example, all persons in a city, where the target of sampling is, for instance, estimating the average age of persons. Those estimators are normally associated with a variance measuring their accuracy [90] .

Sampling is pivotal for most statistical studies for various reasons. For example, obtaining a total population could be purely fictional. For instance, heights of all people in a country. One other potential reason is that processing a whole population census is, more than often, computationally challenging. Despite that this is hardly ever an issue with the abundance of wide spectrum of big data processing engines, at times, it may be true that data arrives in streams where updating results regularly based on newcomers is pivotal for correct time-dependent estimators. In those cases, we usually base our estimates on observations arrived so-far and extrapolate our results to future times. Besides, at times, it's not even practical to visually plot a summary of billions of observations on boards, such as those cases where we generate heat-maps of a natural phenomenon.

Our decision on whether a method is a good or bad sampling method depends highly on various factors including the sampling design and size. The sampling design is the procedure by which a sample of units or sites is selected. However, there is a consensus on the idea that the sample should be a good representative for the population. Stated another way, sample constitutes a scaled-down (can also be dubbed as 'microcosm') version of the population holding intrinsically and mirroring all traits and characteristics of that population it is representing. It is undoubtedly true that there is no such thing like a "perfectly-representative sample", but at least if we could obtain a sample that is good enough to yield characteristic's estimations with a known degree of accuracy or confidence, then it would be safe claiming that the sample is representative. One of the most recurrent problems that renders some

sampling designs as bad is the selection biasedness, which, in simple terms, is the process for which the sampling method overlooks some parts of the population by design [90] . For example, for estimating a percentage of possible voters in the United States who potentially will vote for the democratic party in an upcoming election cycle, selection biasedness may render estimates invalid. It is an indispensable fact that sampling generally cause sampling errors (normally termed as Standard Errors (SE)) which stems from basing estimates on a sample rather than the whole population [90]. Modeling uncertainty has strong ties with selecting proper sampling designs. A design that minimizes uncertainty figures, such as standard errors, is plausible more than those with expanded error intervals. In other terms, as long as those values estimated using a sample are close to the real values (i.e., estimated from the total population with no sampling) for some arbitrary number of sampling permutations, the method is considered good, otherwise not.

Aiming at increasing the unbiasedness coupled with the tendency to design methods that yield low-variance estimates in a variety of scenarios, many sampling methods have been designed, among which the two most widely adopted are simple random sampling (SRS) , which is a probability design (a.k.a. random sampling without replacement) and Simple Stratified Sampling (SSS). The former proceeds by normally assigning an equal selection probability to each unit in the population, thereafter, assigning labels to each unit and selecting labels randomly until a specific number of distinct units that is equal to the sample size is selected. This guarantees that all possible permutations have equal probabilities of being considered as a sample. The latter operates in a different way, where it selects fractional portions from total units depending on the group they belong to. Sampling students from schools, we take 50% boys and 50% girls, where boys and girls are stratum in this case. The distinction between those two magnets lies in the fact that SSS may assign equal inclusion probabilities to each unit in the same stratum, but this may differ from other units in other stratum as each stratum is treated independently [91].

The overarching traits offered by stratification has encouraged us to consider a design that is based on stratified sampling, but at the same time considers the spatial patchy distributions in scenarios of smart cities and Industry 4.0. In the next subsections, we provide a short

primer that spots the light on spatial sampling, aiming at steering a better comprehension for the hybridization we have performed in our method, discussed in section 5.3.4.

5.2.2.2 Sampling

Deterministic solutions for data analytics problems do not play well with fast arriving huge data streams that are mostly geo-referenced with complex data structures that show oscillation in data arrival rates and skewness [4]. Be that as it may, in geo-statistics, approximations that yield plausible error-bounded statistical results are acceptable [92]. Having said that, a well-selected representative sample can be safely exploited for geostatistical analytics such as the approximation of target study variables (e.g., '*average'*, '*total'* and '*proportion'*). Also, observing all items of a population could be intractable, such as observing migrating birds in a huge location, which are spatially unevenly distributed [93].

5.2.2.3 Spatial Online Sampling Designs

Spatial sampling has a great advantage in many domains such as environmental monitoring [94] . It is formally expressed with a ternary $(\psi, \mathfrak{I}, \mathfrak{R})$, where $\mathfrak{R}$ is the embedding space (often two- or three-dimensional space) from which samples are drawn, $\mathfrak{I}$ is the sampling frame (i.e., SRS, SSS) overlaying the survey area (i.e., embedding space), ψ is the statistic that is employed for estimating a variable of interest (e.g., '*total*' and '*mean*' of a parameter in study area). The choices of $\mathfrak{I}$ and ψ heavily affects the goodness of the spatial sampling design [94]. Those configurations enforce an uncertainty on the spatial sample estimation and the common goal is to reach an unbiased estimation with the lowest possible variance, which, in spatial distribution, is normally achieved by being attuned to the characteristics of the spatial data, where the sample is spatially representative and well-spread out over the sampling space [95] .

Preserving spatial co-locality through a sampling design is known to yield better estimates [96, 97]. A principle that complies with Tobler's first law of geography, which simply states that nearby spatial objects are more related than those far apart [98]. One way for achieving this, is to imagine the earth flattened out (i.e., two-dimensional planar irregular grid-like representation) and sample proportional quantities from each subregion (i.e., cell or polygon), which is known to yield plausible statistical results with reduced estimation errors [94, 98] .

Current SPEs with their related spatial-aware extensions and plugins focus on striking a weighted balance between few QoS goals (e.g., low-latency and high-accuracy) by either overprovisioning resources (i.e., scaling in/out) or dropping-off (a.k.a. sampling or shedding) portions from the arriving data, thus loosing tiny accuracy for plausible latency gains. However, overprovisioning resources, that are not normally released after a spike, conflicts with the target of high resources utilization. For sampling and other sketching methods, state-of-art SPEs exploit sampling schemes that are basically embracing randomness, based mostly on SRS [90] , rendering them non-attuned for spatial characteristics that surround objects in proximate locations. SRS does not serve the *estimation quality* QoS target in spatial patchy environments, where spatial objects are normally clumped into few patches. Stated in other terms, SRS normally unduly chooses random counts with unfair fractions from all cells of the survey area (analogous to strata in stratified sampling), even if it performs well at times, at most times it cannot. There is a consensus in geo-statistics that geo-near spatial objects have, more than often, strong ties with contexts of their surroundings (i.e., ecological, anthropogony, etc.,) [93, 99, 100]. All in all, selecting geographically spread-out samples is known to affect estimations quality. We dub those samples drawn that way as *geospatially representative samples*. In addition, although some works of the related art focus on spatially representative sampling designs, they normally consider only static finite populations (as opposed to continuous infinite populations that always have superpopulations). Chief among factors that played a role in the shortage of spatially representative sampling designs for continuous populations is maybe the prohibitive computational capacities of systems at those times. However, current SPEs act as promising jumping off systems for building online sampling designs.

In this thesis, we scope ourselves to designing stratified-alike spatial sampling methods that select well-spread out proportional spatial samples from irregular regions in the sampling space (a.k.a. polygons). It should be also noticed that there are requirements that affect the fact that we are constrained to selecting spatial samples in non-stationary, anisotropy online settings with temporal fluctuations in arrival rates and skewness, thus the term stream sampling (a.k.a. online sampling), which is discussed in the next subsection.

5.2.2.4 Stream Sampling (a.k.a. Online Sampling)

There are requirements that are normally imposed on stream sampling in a way that does not affect finite sampling designs. One important consideration is that samples would be taken either on-the-fly in case of record-at-a-time stream processing models, or from small batches (known as micro-batches) in micro-batch processing models. Another fact is that streaming systems normally apply the exactly-once semantics, where tuples are not replayable. Also, estimates should be designed so that they confluence with the incrementalization semantics of the streaming model. For example, in time-based micro-batching window semantics, an '*average'* on an interesting variable should be updated in every interval (i.e., batch interval, portion of the time window) incrementally building on preceding intervals. Those challenges place many constraints on stream sampling designs that do not normally affect stationary sampling designs in the same way. To close those gaps, we have designed a spatial aware online sampling method that is based on a SPE that supports a declarative SQL-like API. Our system that we term SpatialSPE is discussed in the next section.

5.3 SpatialSPE: QoS-aware Approximate Spatial Data Stream Processing Engine

5.3.1 Usage Model and Baseline System

Intelligent systems such as those focusing on spatial intelligence (refer to section 3.1 for details) serve interactive results from a spatially patchy streaming source to the decision makers in a simplified way that enables them to make sound decisions and strategic plans easily. To achieve that, results are served on the form of either visualizations (e.g., heatmaps, histograms, etc.,) or dashboards. Both are end results that pass through an end-to-end complex pipeline. Map rendering systems are space-constrained in the language of their ability to absorb a limited count of spatial objects and overlay them on a map at any given time. Consider an example of an online spatial query that asks to interactively generate heatmaps of "people and vehicles in-motion grouped by district in the city of Milano in Italy". In a rush hour, were objects are usually clumped into specific districts, this easily cause a clutter. Sampling in this case lends itself as a promising solution. A baseline system that depends on SRS (SpSS-based SRS baseline) normally unduly overlooks regions, resulting in maps that do not necessarily represent the real distributions, which does not help in assisting a correct decision making. In that case, selecting a geospatially well spread-out sample based on a spatial aware design that yields better heatmaps plots. This usage model

convinces the need for an online spatial sampling design. For this, we have designed SAOS as explained in section 5.3.4.

5.3.2 Design Assumptions

To resolve challenges associated with traditional sampling designs such as the case of heatmaps generation, we have designed a spatial aware approximate interactive real-time processing system that we dub as SpatialSPE (an abbreviation for spatial stream processing engine) so that it operates under the following assumptions. Sampling rates are served to the system as an external input, we are not providing any cost model that feeds a controller for mapping QoS goals into an adaptive sampling rate. We have designed instead that kind of a controller as part of the SpatialSSJP (the topic of chapter 6).

5.3.3 SpatialSPE Design Overview

SpatialSPE can be effectively exploited for online spatial approximate analytics. The context diagram of figure 5.1 shows a high-level architecture of the workflow of SpatialSPE. The operation proceeds as follows. Geo-referenced spatial streaming data is fed to the system through an ingestion system (e.g., Kafka) as an unbounded input table (in SpSS terms) at regular time intervals (known as trigger intervals in SpSS terms). SpatialSPE receives the online spatial query in addition to QoS goals (expressed as estimation quality, latency and throughput targets).

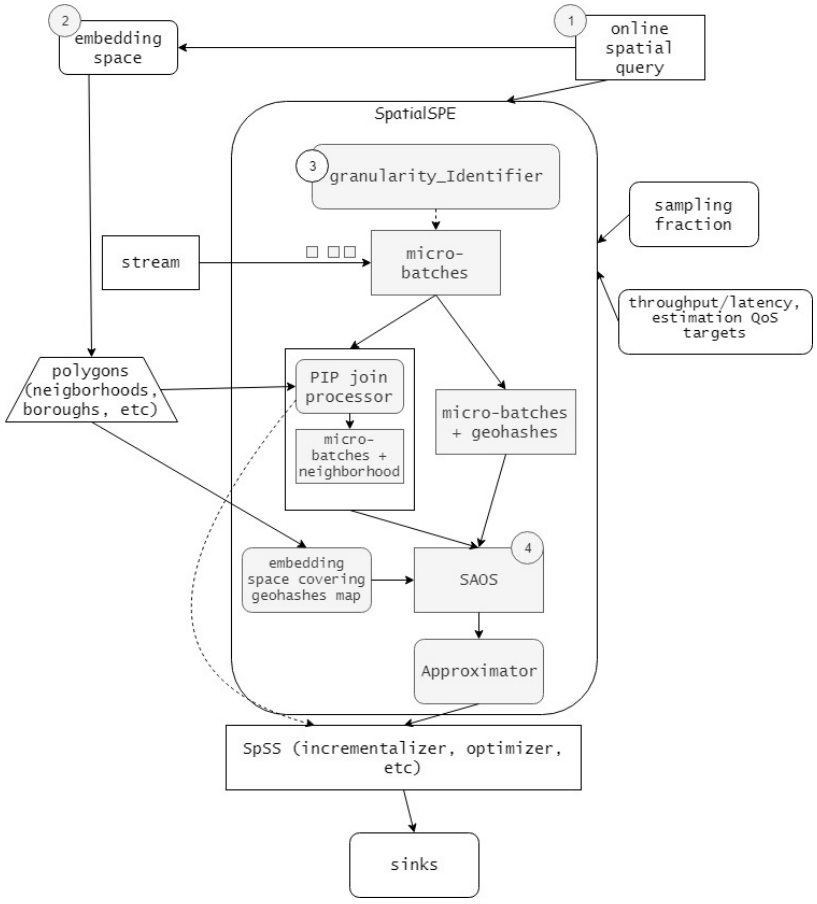

Figure 5.1. SpatialSPE workflow

It also receives a sampling rate (e.g., calculated through an external controller). Listing 5.1 shows an example online query (in Spark SQL terms)

```
df = samplepointDF_SSS.groupBy($"geohash"). count().orderBy($"count".desc)
```

listing 5.1 An example online query in Spark Structured Streaming terms

SpatialSPE *granularity identifier* (a building block in SpatialSPE, see figure 5.1) decides the level of granularity to apply (most granular level is geohash, while the coarser level has no limits, could be borough, district, neighborhood, county in city administration terms, or even cities, countries, etc.,). In cases where the most granular level is required, for example "sampling fairly proportional amounts based on a grid-like representation", imagining the

earth in two planar geometry with each grid cell as a covering for a circle (squared-grid with width equals to double the radius), then the *granularity identifier* forwards micro-batches untouched to a component that simply adds geohashes to each unit (a linear dimensionality reduction approach, from GPS coordinates, longitude/latitude, into a geohash code). On the contrary, in cases where a coarser level is requested, such as sampling by 'borough' instead of 'geohashes', thus taking evenly proportional sampling rates from each borough in a city, the problem is more complex and needs more attention. First, each spatial point in the micro-batch can be converted to a geohash, then specifying to which borough this point belongs requires solving the Point in Polygon test (recall more information from section 2.3.2), which requires joining data from each micro-batch with polygons table (a static table, where each polygon represents a borough). To solve this problem efficiently, we have employed a retrofitted version of Spark's Magellan [9] (a geospatial analytics library that was designed to work with Spark SQL). We have chosen Magellan because it supports SQL and it is known in performing a cheap PIP tests by utilizing filter-and-refinement join approach (that employs a cheap MBR-join in the filter stage). Hitting this point, spatial objects are readily stratified and will be fed to our spatial aware online sampling algorithm (abbreviated SAOS, explained shortly in section 5.3.4). SAOS selects fairly proportional amounts (based on the granularity level identified) from all regions and serves the resulting sample to an approximator that operates on top of SpSS , taking advantage of the incrementalizer and optimizers of the underlying system in generating incremental query results(e.g., every time window) .

SpatialSPE operates under the assumption that QoS goals (e.g., latency/throughput) are served as an input by the user. A model-based function is responsible for calculating those parameters into an appropriate sampling fraction (i.e., rate), which is also an external module. SAOS then samples proportional fractions from the data input stream, this is followed by an application for the retrofitted incrementalizer, which computes the geo-statistics from the samples and produce a result with rigorous error bounds and serve them to user interactively.

[9] https://github.com/harsha2010/magellan

At this stage, sampling ratios (i.e., fractions) are the same for all stratum (i.e., geohashes). Also, the system receives a CQ that will be executed stepwise.

An important sub-module of SpatialSPE is responsible for calculating the covering geocodes (currently geohashes). Those are the geohashes corresponding to MBRs that are covering an embedded space where the sample has been drawn. This submodule works as follows; it first receives an input file containing coordinates of vertices that are forming polygonal areas (a.k.a. neighborhoods, districts or boroughs in city management terms) covering collectively the survey area. The procedure proceeds by exploding all the geohashes covering all the polygons, thus building a map and serving it to SAOS, which selects a well representative spatial sample and emits it to the operator downstream.

Algorithm 5.1. SpatialSPE Workflow

/ latThrTargets: latency throughput targets, precision: geohash precision, CQ: continuous query*/*

Input: stream, ContinuousQuery (CQ), latThrTargets, polygons, geoPrec, seed

1: samplingMap ← ∅ *//map of geohash keys and sampling fractions*
2: coverGeo ← getCoverGeo (polygons, geoPrec) */* List of geohashes covering study area (embedding space) */*
// costProcedure: external cost model that calculates the sampling fraction
3: sampFraction ← costProcedure(*latThrTargets*)
4: **Foreach** geohash **in** coverGeo **do**
// construct a map, geohash: key, sampling fraction: value
5: element ← map {geohash→ sampFraction}
6: samplingMap.put(element)
7: **End**
8: **Foreach** time window interval **do**
9: windowSample = ∅ *// tuples sampled in current time window*
10: **Foreach** batchInterval **in** window interval **do**
11: batchSample = ∅ *//tuples sampled in current batch interval*
12: **forall** $tuples_i$ **in** batch interval **do**
/ apply SAOS on tuples of current batch interval: $tuples_i$ */*
13: batchSample ← SAOS ($tuples_i$, samplingMap, sampFraction, seed)
14: windowSample.add(batchSample)
15: **End**
16: **End**
//compute and serve incremental output after each time window
17: ***incrementalOutput*** ← run (CQ, windowSample)
18: **return** incrementalOutput with error-bounds
19: **End**

The operator is part of a DAG that is corresponding to the user-defined CQ, which then will be applied to the sample during that time window interval and the result will be interactively served to user incrementally, reflecting inter-window changes. Algorithm 5.1 shows the workflow of SpatialSPE. For more information about Algorithm 5.1, refer to our paper [101].

5.3.4 Spatial Aware Online Sampling (SAOS) Algorithm

To enable SpatialSPE in achieving QoS goals for spatial real-time data analytics scenarios that require approximations, we have designed a unique sampling method that we dub as Spatial-Aware Online Sampling (short for SAOS), comprising a pivotal technological block in our system SpatialSPE. Our algorithm is superior because of its unique ability in considering spatial patchy distributions by being able to collect fairly proportional amounts without overlooking some sampling areas. SAOS seamlessly and transparently is incorporated within the layers of a de facto micro-batch-based SPE (specifically SpSS). Thus, captivating advantages of the incrementalizers and query optimizers of the underlying engine. SAOS does not necessitate a prior-knowledge of streaming statistics (e.g., total data population, where even such a semantic vanishes as any population in continuous settings is part of a superpopulation).

The workflow of SAOS is listed in Algorithm 5.2. It proceeds as follows; during each trigger interval within a tumbling time window, for micro-batches (tagged with geohashes or coarser containment polygon such as ‘neighborhood’) SAOS refers to a fraction map to read the corresponding sampling fraction for each stratum (i.e., geohash, neighborhood, etc.,), then it applies SRS to each stratum independently to select a count that is equal to the fraction specified, such that each point within each stratum (i.e., geohash) has an equal inclusion probability. For an explanatory utilitarian perspective, SAOS algorithm resembles a heuristic overview such as follows. Considering the earth flattened out to a two-dimensional planar space, we first overlay a square grid on the embedding space (i.e., the space where samples are drawn), where SAOS design frame resorts to a recursive halving in one-dimension and quartering in two dimensions. This is in case of a stratification based on geohash, whereas on a coarser level the grid is irregular. Thereafter SAOS selects randomly a proportional number of spatial objects from each grid cell independently (or from each borough, district,

etc., in case of a coarser stratification level). By this design, we recover stratified sampling, which is plausible in geo-statistics. Changing the precision of geohash affects the number of covering cells in the survey area, which allows user to control the system in a drill-down/roll-up fashion. Such a hybridization between sequence ordering (i.e., geohash imposed on a grid-based hierarchal representation) with SRS (imposed within each grid cell independently) yields a geospatially well-representative sample, which is known to result in better estimation quality in geo-statistics. For more information about that refer to our paper [101].

Algorithm 5.2 Spatial-Aware Online Sampling (SAOS)

```
1: SAOS (tuples_i, samplingMap, sampFraction, seed)
2: r = random(seed), S ← ∅
3: Foreach tuple in micro-batch-tuples do
4:    geohash ← geocode (tuple)
      //get sampling fraction for this geohash key = fraction_i, or zero
5:    fraction_i ← samplingMap.getOrElse(geohash,0.0)
      //toss a coin for selecting items from each geohash in current batch
6:    If (P (r < fraction_i)) S.put(tuple)
7: End
8: return S //return a set S containing the sample
9: End
```

5.3.5 Spatial Queries Supported

Along the lines with the design goals that we have stipulated for SpatialDSMS (refer to section 3.4.1), SpatialSPE currently supports two primitives of spatial queries; single and ensembles, which then can be used seamlessly to *compose* other more advanced queries (e.g., spatial clustering [102] and spatial online clustering). Recap that composability is one of the design goals of SpatialDSMS. We rely on the theory of stratified sampling and the theory of random sampling [90] for approximating spatial queries in SQG1, some equations in this section are adapted from [90].

Spatial Queries Group1 (SQG1). **Single spatial queries** (i.e., **linear**). An example spatial query in this category is an interactive request to "find the average trip distance travelled by taxis originating from a specific district in a metropolitan city". Because SAOS resorts to a stratified-like sampling design, we depend on the theory of stratified sampling for

estimations (e.g., '*means'*, '*totals'*, etc.,) [90]. Having said that, estimating the '*average'* envisaged in the query can be formalized as follows. Imagine that we have *K* geohashes in total (each geohash overlays a stratum, imagining both as grid cells), y_{kj} is a value of a j_{th} tuple in geohash *k*, then t (pronounced *tau*) is a population '*total'* for stratum *k*, which follows that a population '*total'* for the target parameter y is estimated by SAOS through applying the formula in (5.1).

$$\hat{t}_{SAOS} = \sum_{k=1}^{K} t_k = \sum_{k=1}^{K} N_k \bar{y}_k \tag{5.1}$$

Then using SAOS the average is estimated by applying (5.2).

$$\bar{Y}_{SAOS} = \hat{t}_{SAOS}/N = \sum_{i=1}^{I} (N_i/N)\bar{y}_i \tag{5.2}$$

Where $\hat{t}_{SAOS}$ is the estimated '*total'* by applying SAOS, N is the number of tuples received thus far, N_i is the number of tuples received heretofore in stratum *i*, $\bar{y}_i$ is the incremental '*average'* in stratum *i* calculated up to now.

For SpSS-based SRS baseline, we first apply (5.3), to estimate the '*mean'*

$$\bar{Y}_{SRS} = \Sigma_{k \in SRS} y_k / n \tag{5.3}$$

where y_i are the values of target variables in every time window, n is the size of the sample in every time window.

SpSS does not natively support those estimators, we have incorporated a glue specifically for incrementalizing those estimators, taking advantage of the incrementalizer (a building block in SpSS) provided by the Spark engine. A query in this category is similar to the one in listing 5.2, which is asking to "calculate the '*average'* trip distance travelled through all taxi trips in NY City, USA every minute"

```
data.where("city = NY").groupBy(window("time","60
seconds").avg("trip_distance")
```

listing 5.2. average statistic estimation spatial query example in Spark terms

Spatial Queries Group2 (SQG2). In this group falls **stateful spatial online aggregation queries** (i.e., **ensembles**). Online aggregations differ from static batch counterpart in that the former requires managing state between batch intervals, thus achieving a consistency. In this thesis, we focus on Top-N (a.k.a. top-K) online aggregations. SAOS is applied to arriving spatial points , thereafter they are grouped by geohash keys (Also it is possible to group on a coarser level such as neighborhoods, boroughs, or districts), and then a count predicate is applied calculating tuples number for every geohash incrementally and a sorting function is applied in a descending style. An example spatial query belonging to this category is the follows. “which are the top-10 boroughs in NYC where people tend to order green taxi pickups”. Listing 5.3 shows this query expressed in Spark SQL terms.

```
val sampleStatistics = sample
    .groupBy($"borough ", window($"time", "1 minute"))
    .count().orderBy($"count".desc)
val query = sampleStatistics.writeStream
  .queryName("statistics")…start()
statistics.select($"borough",$"count").limit(10)
```

listing 5.3. Top-N spatial query example

5.3.6 Quantifying the Uncertainty Associated with Sampling

Estimating target variables by sampling instead of the population is naturally bounded to an uncertainty which should be quantified to measure the ability of the sampling design in achieving the QoS goals predefined by the user. Since SAOS in its core resorts to stratified-alike sampling, then the theory of stratification applies. We rely on the theory of stratified sampling and the theory of random sampling [90] for quantifying the uncertainty of applying spatial queries in SQG1 to estimate target variables, some equations in this section are adapted from [90].

I) For ***SQG1*** **(single spatial queries)**, since SAOS recovers a stratified-alike sampling design, we depend on the Theory of Stratified Sampling [90] for producing statistically acceptable estimations of the accuracy of approximations

for ***SQG1*** queries that are obtained by applying SAOS instead of a SRS. We first apply (5.4).

$$\hat{v}(\hat{t}_{SAOS}) = \sum_{k=1}^{K} (N_k - n_k/N_k)\ (N_k^2 s_k^2/n_k) \qquad 5.4)$$

Where n_k is the number of tuples thus far in stratum *k*, N_k is the total number of items up to now in all strata, s_K^2 is the standard deviation in stratum *k*. All those magnitudes are calculated incrementally by our support.

In order to compute an estimated variance for the estimated total. Then we incorporate the result in an equation to estimate a variance for the estimated average of the target variable, specifically by applying (5.5).

$$\hat{v}(\overline{Y}_{SAOS}) = \hat{v}(\hat{t}_{SAOS})/N^2 \qquad 5.5)$$

Where $\hat{v}(\overline{Y}_{SAOS})$ is the estimated variance of the estimated mean, $\hat{v}(\hat{t}_{SAOS})$ is the estimated variance of the estimated total.

Thereafter, we compute standard error (SE) depending on (5.6).

$$SE(\overline{Y}_{SAOS}) = \sqrt{\hat{v}(\overline{Y}_{SAOS})} \qquad (5.6)$$

Then we carry the value obtained of SE and apply it in (5.7).

$$\overline{Y}_{SAOS} \mp z_{\alpha/2} SE(\overline{Y}_{SAOS}) \qquad (5.7)$$

In order to approximate $100(1-\alpha)\%$ confidence interval (CI) of the population mean $\overline{Y}_{pop}$, where $z_{\alpha/2}$ is the upper $\alpha/2$ point of normal distribution. Thereafter we define relative error as in (5.8). SE measures sampling distribution variability (not to be confused with standard deviation, which measures the variability on points level).

$$RE = z_{\alpha/2}(SE(\overline{Y}_{SAOS})/\overline{Y}_{SAOS}) \qquad 5.8)$$

The intuition behind this adjusted error metric is that values of SE metric are normally small, so we have used a relative error as a representative that preserves the same SE trend but being more meaningful. We also define an accuracy loss by (5.9).

$$\text{accLoss} = |\text{estimatedMean} - \text{trueMean}| \,/\, \text{trueMean} \tag{5.9}$$

We also define the gain by applying SAOS instead of the SRS-based baseline, for which we apply (5.10).

$$\text{gain}_{SAOS} = \hat{v}(\bar{Y}_{SAOS}) \,/ \hat{v}(\bar{Y}_{SRS}) \tag{5.10}$$

, where $\hat{v}(\bar{Y}_{SAOS})$ is the estimated variance resulted by applying SAOS, whereas $\hat{v}(\bar{Y}_{SRS})$ is the estimated variance resulted by applying SpSS-based SRS baseline.

Also, we apply the following equations from the theory of SRS to calculate the estimated variance estimated average and other quantities. Then we apply (5.11) to calculate the estimated variance of the estimated mean.

$$\hat{V}(\bar{Y}_{SRS}) = ((N - n/N)(s^2/n) \tag{5.11}$$

Where N is the total number of records arrived at the system at the time of computation, s^2 is the incrementalized variance calculated from the sample drawn thus far.

Then we apply (5.12) to calculate the standard error

$$\text{SE}(\bar{Y}_{SRS}) = \sqrt{\hat{V}(\bar{Y}_{SRS})} \tag{5.12}$$

Then we apply (5.13) to calculate a relative error.

$$\text{RE} = z_{\alpha/2}(\text{SE}(\bar{Y}_{SRS})/\bar{Y}_{SRS}) \tag{5.13}$$

For the same rationale that we have suggested beforehand regarding the relative error in SAOS case.

II) For ***Spatial Queries Group2 (SQG2),*** **online spatial stateful aggregations** (specifically Top-K) queries. We measure every method ability in preserving an original ranking that would be obtained if we have access to a population or a superpopulation. This is due to the fact that the online stateful aggregations we

compute by applying sampling instead of the population depending on a baseline normally has a quality guarantees in terms of accuracy. To measure the ability in satisfying those qualities, we apply a Spearman's rank correlation coefficient [103] (read Spearman's *rho* hereafter). We have retrofitted the measure so that it applies to our case. Spearman's *rho* is a measure for statistical dependency between the ranking of two variables in a dataset. In short, our application of *rho* proceeds as follows. We collect the ranks (i.e., orderings), and once the spatial CQ stops (i.e., shutdown by user, or depending on a query window semantics) we take the collected orderings of the original aggregations (i.e., those that would result from a population without sampling, we consider the total number of tuples emitted by the sources at that point as the population) and the ranking that is calculated by applying SAOS (and in the same vein, by applying SpSS-based SRS baseline). Then we serve those figures to Spearman's *rho* and apply (5.14) accordingly.

$$\rho_{rg} = \text{covariance}(\text{rank}_{\text{nosampling}}, \text{rank}_{\text{sampling}}) \,/\, (\sigma_{\text{rank}_{\text{nosampling}}} \,.\, \sigma_{\text{rank}_{\text{sampling}}}) \quad (5.14)$$

, where ρ_{rg} (i.e., *rho*) is spearman's correlation coefficient applied for ranking statistics , $\text{covariance}(\text{rank}_{\text{nosampling}}, \text{rank}_{\text{sampling}})$ is the covariance of the rank variables, $\sigma_{rank_{\text{nosampling}}}$ and $\sigma_{rank_{\text{sampling}}}$ are the standard deviations of the rank variables, without and with sampling, respectively.

5.4 SpatialSPE Implementation Technical Details

To show that SpatialSPE[10] is adept and versed in achieving QoS goals, specifically time-based qualities such as high *throughput* and low *latency*, in addition to accuracy-based QoS goals, specifically the *estimation quality*, we have implemented a standard-compliant prototype based on SpSS, stacking up our patches on SpSS. Because, as of the time of this writing, SRS is not implemented in SpSS, aiming at an equal comparison, we have

[10] The source code of SpatialSPE (together with SAOS sampling method) is available at: https://github.com/IsamAljawarneh/SpatialSPE

incorporated a patch that glues a version of an online SRS , which can operate in streaming settings for approximating estimators depending on SRS (we dub this version as *SpSS-based SRS*). The micro-batching model of SpSS has enabled us to implement this transparently, where source tuples are collected in blocks (i.e., micro-batching mode of operation) before being split for processing. Thus, by injecting a frontstage after the block formation stage and exactly before partitioning, our patch for SRS works as if it is operating in a batch mode (the core concept of micro-batch stream processing).

We did this because we needed a comparable ground-truth with which to compare our SAOS method. The two methods, the baseline SpSS-based SRS and SAOS belong to the family of sampling without replacement.

5.5 Performance Evaluation and Results

5.5.1 Comparison Methodology

To compare with the baseline, since the goals we aim at achieving are novel, to the best of our knowledge, with same settings, including the utilization of a declarative API-based streaming and the incrementalization of statistical estimators in non-stationary spatially-rich environments, we are not aware of any similar system from the literature that is achieving the same goals. We build on top of SpSS, which currently, as of this writing, does not have a native support for sampling on streaming DataFrames/Datasets (core SpSS abstractions). We needed a baseline for which to compare our method with, so we have decided to retrofit SpSS so as to enable incrementalizing an SRS-based method as a patch on top of the stack, and then compared our SpatialSPE with that. The SRS-based method works by simply applying the traditional SRS for every micro-batch without replacement and with equal inclusion probabilities. Replacement is not preferable in our setting because the nature of the data torrents does not guarantee equal chance of inclusion for all arrival tuples in case of replacements. It is often possible that replacement units are also non-response units, thus deteriorating the inclusion probability.

5.5.2 Metrics of Interest

We first define a set of metrics of interest that we apply for the queries that we are supporting (recap them from section 5.3.5) . We depend on two fundamental targets; those are normally found in all AQP systems. In particular, we rely on estimator accuracy (estimation quality,

recap section 3.2) and processing throughput for measuring the QoS in terms of quality of estimates and the ability of the system in keeping up with data skewness and arrival paces, respectively. Also, we measure an additional functional QoS metric, scalability.

Among those metrics, we have measured throughput, latency, accuracy (estimation quality and gain), and scalability for all queries belonging to the single queries group (SQG1) (linear queries that return single values such as '*averages'* and '*totals'* estimator). For stateful aggregations (belonging to group SQG2, refer to section 5.3.5), we measure throughput, latency, estimation quality (through Spearman's *rho*) and scalability. We vary multiple parameters to measure a mixture of those metrics (i.e., trading off a mashup between them), those basically include varying sampling fractions, arrival rates and geohash precisions as detailed in section 5.5.4. We now conceptualize definitions of the foregoing metrics.

Throughput. Refer to Appendix C for technical details explaining the way we compute throughput for SpatialSPE.

Latency. We have calculated it the same way as described in section 3.2.

Estimation quality. We measure the **estimation quality** of single queries of estimators for both, our SpatialSPE and adapted SpSS SRS-based, by utilizing the Standard Error (SE) general formula from the Central Limit Theorem [90] and calculating a relative error. Also, we apply *rho* for measuring the quality of ranking statistics (i.e., the estimation quality of queries in SQG2, refer to section 5.3.5)

Scalability. We define the scalability as the ability of the system to perform either faster (in terms of latency) or keep, at least, with the pace of data fluctuating arrival torrents at moments of transient spikes. For measuring scalability, we vary the parameters of cluster deployment, more specifically, increasing the bare-metals of our cluster from two to four worker nodes (see section 5.5.3 for details of a worker node characteristic), thus doubling computation power and sensing the effect against arrival rates.

We vary the stream arrival rate as this is the case in reality, where streams exhibit oscillating rates over time. Higher stream rate carries a greater number of input units and may cause latency to rocketry climb, thus studying the effect of this on incrementalizations in parallel settings is pivotal.

5.5.3 Experimental Setup and Datasets

5.5.3.1 Dataset

For benchmarking, we use the NY City green taxicab trips datasets [11], where we select a big cohort representing six months (almost nine million tuples) expressing taxis rides for the first half of year 2016. Data includes spatial fields, geometrical planar representations of pick-up/drop-off locations and trip distances measuring the distance travelled for each trip. Despite the skewness of this data, traditional sampling theories applies in accordance with the Central Limit Theorem (CLT) [90]. Refer to Appendix D for further details.

5.5.3.2 Deployment and experimental settings

We deploy SpatialSPE on a Microsoft Azure HDInsight cloud computing 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 nodes 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.

5.5.4 Evaluation Strategy

In this section, we discuss the results we have collected through measuring the QoS metrics that are explained in section 5.5.2. Our evaluation strategy varies four parameter configurations; sampling rate, stream source data arrival rate, geohash size and computing resources size. We intermix those configurations for measuring the QoS metrics by executing queries of section 5.3.5. The two parameter configurations are as follows.

- ***Parameter Configurations #1.*** Varying geohash size and sampling rate. We vary geohash size between 30 and 35. Also, we vary the sampling rate between 20% and 80% (20% step length). By this configuration mashup, we target measuring the *accuracy* QoS goals (i.e., estimation quality) of all query types in groups SQG1 and SQG2.

[11] https://www1.nyc.gov

- ***Parameter Configurations #2***. Fixing geohash size to 30 and varying spatial stream tuples arrival rate from 1 million to 2 million tuples/second, in combination with varying sampling rates between 20% and 80% (with a step size that is equal to 20%), and also including 1% and 5% to account for harsh latency goals in cases where spikes in data arrival rates are brutal. By this configuration mashup, we measure the QoS *throughput* and *latency* goals of spatial queries in SQG2, because they consist of computationally expensive online stateful aggregations.

All measurements are computed as a median (i.e., 50th percentile) of ten running sessions (i.e., repeating same queries with same configurations 10 times and taking the average of the skill).

5.5.5 Test Cases and Results

5.5.5.1 Testing scenarios

1) **SQG1** test cases. We have measured the performance (i.e., the achievement of *estimation quality* QoS goal) by applying the following query (which belongs to SQG1): "what is the accumulative average of a trip distance travelled by taxicabs itinerary trips within first six months of 2016".
2) **SQG2** test cases. For top-N rankings, we apply an online spatial aggregation query, specifically the following; "what is the top 10 neighborhoods (or circular locations bounded by MBRs, geohashes) in NY city, USA where taxicabs trips originate".

5.5.5.2 Results and discussion

5.5.5.2.1 **SQG1** test case results

We use parameter configurations#1 for running those tests in this subsection.

Figure 5.2 elucidates the differences between the online sampling schemes SAOS and SpSS-based SRS in terms of the "estimation quality" of an estimator for a target variable (such as the '*average'* requested through **SQG1** test cases as explained in section 5.5.5.1).

As it is evident, SpSS-based SRS underperforms SAOS in terms of the estimation quality (measured through RE and accuracy loss). Increasing the geohash size negatively affects the estimation quality. Results shown here are measures for confidence interval 68%. The same

pattern applies for the confidence intervals 95% and 99%. Refer to our paper for more interesting results [101] .

Notice that, despite seems trivial, relative errors signifies an important aspect in regard to estimation quality. To make sense of it, reducing the error by a factor of 2 requires at least a sample that is bigger by a factor of 4. This means that even a small fractional gain in terms of those measures (i.e., accuracy loss and relative error) significantly meets the accuracy QoS goals (i.e., higher estimation quality).

To take a more utilitarian perspective of how this effect (even looks small in figures) can negatively impact the estimation, we show in figures 5.3 and 5.4 , respectively, how by using SpSS-based SRS the estimator misses the 68% confidence interval (for the mean estimator) at some sampling rates, whereas SAOS is perfectly fitting within the boundaries of the same CI. The same trend occurs for 99 and 95 confidence intervals, refer to our paper for more interesting results [101].

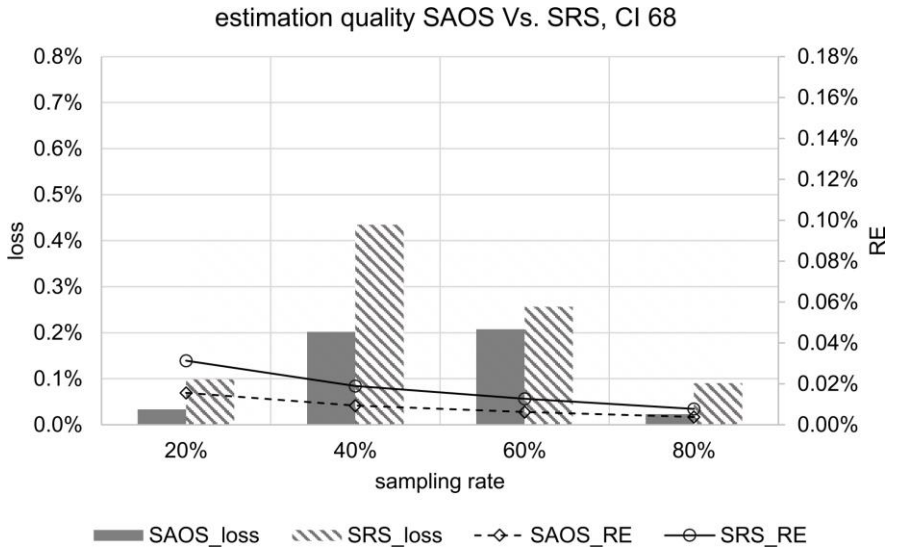

Figure 5.2. Estimation accuracy of SAOS vs. SpSS-based SRS, for G1 queries. 'loss' in the legend is the accuracy loss calculated by applying equation (5.9), whereas 'RE' is the relative error calculated through equations (5.8) and (5.13) for SAOS and SpSS-based SRS, respectively

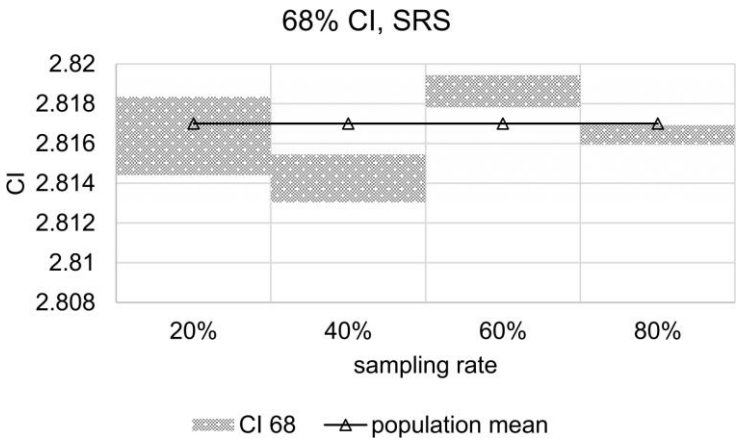

Figure 5.3. CI 68% SRS on mean estimator varying the sampling fraction. CI in the legend is the confidence interval

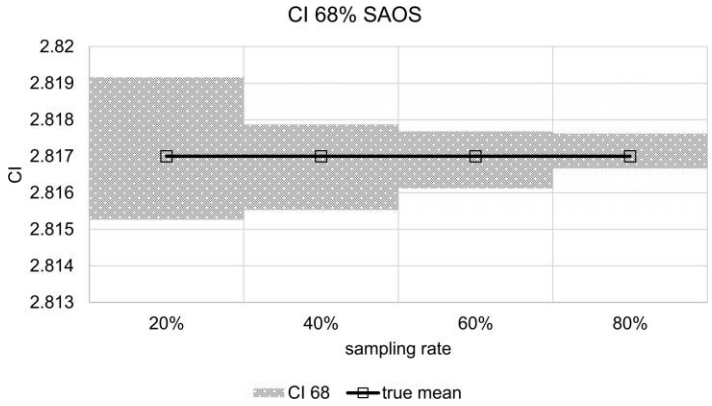

Figure 5.4. CI 68% SAOS on mean estimator varying the sampling fraction. CI in the legend is the confidence interval

To better understand how SAOS is adept more than SRS in geo-statistics, we show in figure 5.5 the gain obtained by applying our method to **SQG1** queries (calculated by applying the design effect measurement [90] , refer to section 5.3.6).

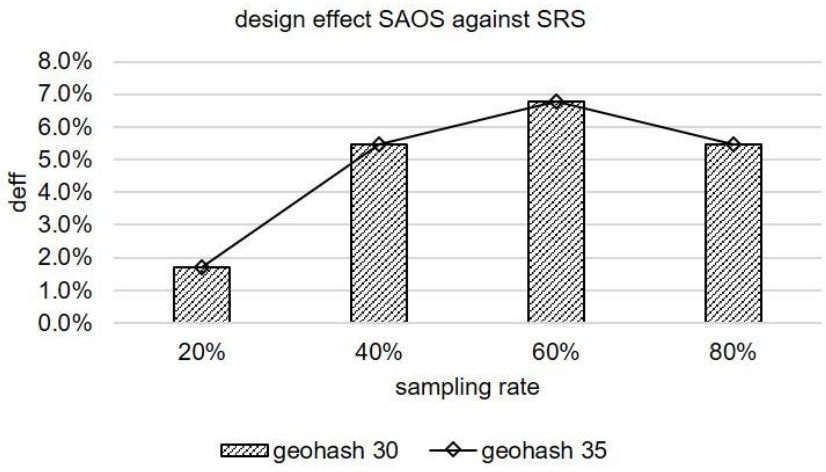

Figure 5.5. design effect by applying SAOS against SpSS-based SRS

Notice that we obtain as large as 7% gain by applying SAOS against SpSS-based SRS. If these figures were the population variances, we would expect that we would need on average only (1000k).(0.93) = 930k observations with a sample from SAOS to obtain the same *estimation quality* as from an SRS of 1000k observations, this saves (70K tuples less, for an arrival rate of 1 million tuples/second, this means that we take 70 thousands tuples less, which is statistically significant) a precious time of online processing in latency-sensitive SPEs, where even milliseconds can save the system from coming into a halt.

5.5.5.2.2 **SQG2** test case results

Figure 5.6 depicts the skill of SAOS in comparison to the baselines (SpSS-based SRS) in the language of *estimation quality* for spatial queries of SQG2, where Top-N ranking quality of SAOS outperforms SRS, despite almost at par for some sampling rates and geohash sizes.

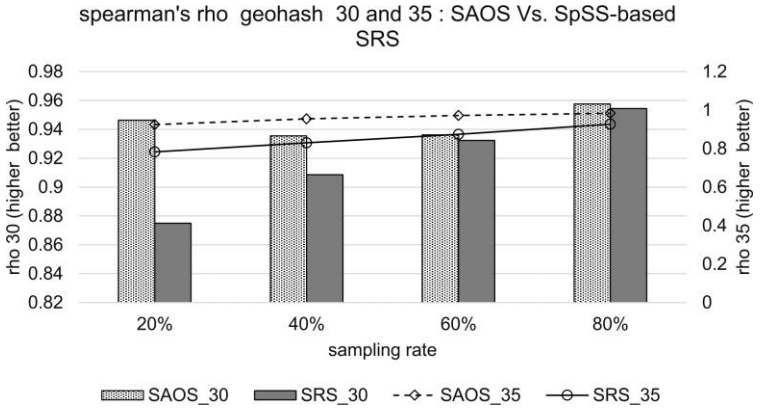

Figure 5.6. Spearmans's *rho* by applying SAOS Vs. SpSS SRS-based. 'rho 30' (in the primary access) means *rho* value at geohash precision 30, whereas 'rho 35' (in the secondary axis) means *rho* value at geohash precision 35

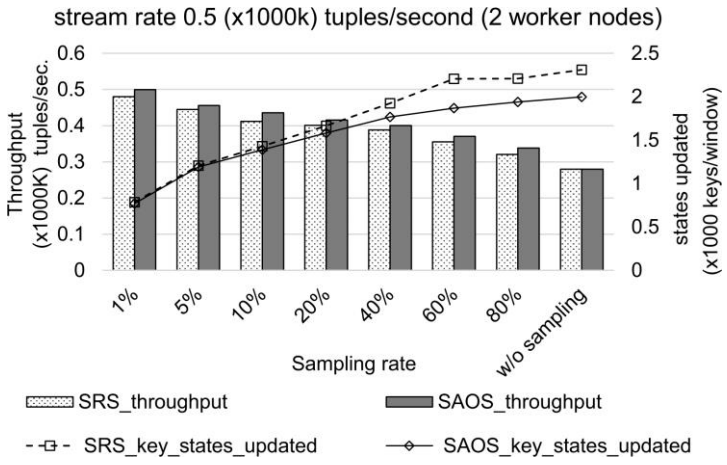

Figure 5.7. Throughput by running SAOS against SpSS-based SRS, with a streaming rate that is equal to 500k tuples/second. 'key_states_updated' (in the secondary access) in the legend means the average number of keys updated between tumbling windows

Speaking about time-based QoS goals (more specifically the throughput in this case), Figure 5.7 elucidates that SpSS-based SRS slightly underperforms SAOS. Despite being a simple approach, SRS in this case performs worst because, on average, the system needs to manage more key states between triggers when applying SRS, this is basically due to the fact that

SRS is not aware of keys distribution and selects tuples totally randomly, which means sampling unnecessarily more keys every trigger.

SAOS and SpSS-based SRS act in the same way for data oscillation from 500K to 1000K to 2000K tuples/second, while SpSS-based SRS always underperforming. Refer to our paper for more interesting results, specifically those showing the same latency and throughput trends on different settings (four worker nodes instead of two) [101]. All in all, SAOS is able to handle the pace at which data is arriving (almost at the par), thus achieving the *latency* quality goals.

We finally show the effect of incrementalization on mean estimator. Figure 5.8 shows how both SAOS and SpSS-based SRS are catching up with the true mean value after a total number of a million tuples arrived. The *mean estimation* for both is approaching stepwise the true value. However, SAOS is approaching faster and this is further self-explained by the smaller value of standard error that is resulting from applying SAOS (as opposed to the value obtained by applying SRS), calculated incrementally. Notice also however, that the *standard error (SE)* difference for both converges and vanishes as their estimates approach the true value.

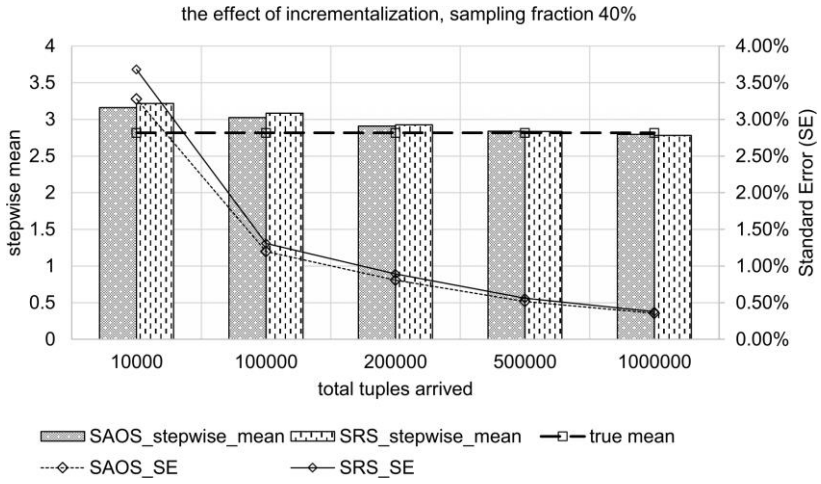

Figure 5.8. the effect of incrementalization on the 'average' or 'mean' estimator. Sampling fraction is set to 40 %. In the legend, 'stepwise_mean' (the primary access on the left) is the 'mean' value changes in correspondence to total tuples arrived up until that point in time. SE (the secondary access on the right) is the standard error.

In theory also, SAOS is an appealing and compelling approach, a theoretical perspective explaining the excellence of SAOS is explained in Appendix E.

5.6 Similar Works

In relevant literature [99, 100, 104] apply various dimensionality reduction approaches, but however are computationally expensive and inapplicable in distributed online deployments. Also, relevant art of the literature focus on achieving single QoS goals (for example, satisfying either high-resource utilization or low-latency) without seeking a balanced weight between them.

Several works can be traced in the literature focusing on spatial sampling. However, most of them are geared toward centralized and stationary settings, depending on High Performance Computing (HPC) deployments with disk-resident datasets. While this works for some scenarios, it was not usually the case during the last decade, where spatially-augmented huge data amounts are arriving very fast, with sometimes burst loads and unruly spikes (i.e., not amenable to discipline), thus leading to an interest in online spatial sampling. This is specifically challenging, giving that implanting spatial awareness normally presents systems with additional overheads, due in part to the ‘curse of dimensionality’ of geospatial objects representations.

Most relevantly, [105] have designed a dimensionality reduction method for finite populations, dubbed as generalized random-tessellation stratified (GRTS) , that is based on mapping two-dimensional into lower-dimensional space, then creating a set of randomly ordered spatial addresses with a mix of systematic sampling in order to generate a well-balanced random sample. They depend on the fact that spatial objects that are proximate in the two-dimensional planar space tend to be proximate in a lower one-dimensional space after mapping. The sample is then selected using a systematic sampling scheme. This is analogous to random tessellation in a two-dimensional space. However, well-spread does not necessarily mean well representativeness and the systematic component may under-represent some regions. In the same vein, [106] presents a sampling method that relies on dimensionality reduction, more specifically by utilizing space-filling curves. They order the survey units in such a way that consecutively numbered points represent spatially well-

balanced sample. Other works include [96] which has incorporated kriging estimator for a real-time monitoring of environmental phenomena into a higher-level architectural pattern.

The picture that emerges from the relevant literature, however, is that, none of the forgoing studies are applicable in distributed deployments. Hence, they are not designed to achieve incrementally accurate results that improve dynamically over time (i.e., stepwise). On the contrary, our system was adept in achieving spatial-awareness in distributed settings. Also, SpatialSPE has introduced incrementalization over geo-referenced data streams using a declarative API, a target that is completely novel.

5.7 Chapter Conclusion and Forward

The idea that spatially-balanced sampled datasets yield better estimations than simple probability sampling methods is well established in the relevant literature. In accordance with that, there are some frameworks for incorporating spatial awareness into statistical sampling. Some methods are based on splitting the study area into cells (traditionally known as tessellation, which implies dividing the study area into polygons, either equally- or arbitrarily-sized) and treating each cell as a stratum, thus simplifying the application of stratified-alike sampling designs, which is plausible in geo-statistics. However, those methods are not ready for distributed computing settings. Furthermore, they incorporate computationally expensive structures, such as tree-based hierarchal representation structures that renders them, despite being efficient theoretically, unsuitable for extension to the distributed computing world.

On the other side, distributed big data processing systems are evolving fast in an unprecedented way, reflecting the need for systems that adapt to the fluctuating and oscillating pace of big datasets that show temporal skewness.

In this chapter, we have shown SpatialSPE which constitutes an integral building block of our system SpatialDSMS. It is complementing the accurate computations (performed through SpatialBPE and SpatialNoSQL, topics of chapter 4) for scenarios that need approximations to be performed on fast arriving online spatial torrents of data loads. SpSS does not have a native support for SAQP and SpatialSPE is transparently incorporated with SpSS and complementing it in that dimension. Again, we have achieved one of our design goals, specifically the modularity, in the way that we have designed SpatialSPE so that it

accepts any future patches for SAQP. The novelty of our system is that it is the first-in-class that is being built on top of an SPE that offers a declarative API, thus naturally offering an interface that allows expressing geometric approximate computations in a human-friendly manner and as a batch-like query as if it was to be executed in batch mode, while underneath reusing optimizations provided by such a promising SPE.

Because of being simple and conceptually appealing approach, the desire by statisticians to employ SRS is high. However, this does come at the inconvenience of poor results obtained in geostatistical settings and it suffers from computational limitations, and we advise that in those settings it should only be envisioned as a last resort, and there could be a qualitative leap between using SRS and SAOS. This does not however detract from the value of SRS, but otherwise complementing it and extending its usefulness to the current world that is rich of spatial data. Beyond its theoretical impact, our method performs the best in the wild and as a hybridization between SRS-based and SSS designs, it is the best-of-both-worlds retaining benefits of both without their drawbacks.

SAOS was able to incorporate seamlessly and transparently within the layers of the semantic representations of an emerging declarative micro-batch streaming model, yielding statistically significant estimates with rigorous error bounds. So far, we have considered sampling fractions that are served to SpatialSPE by an expert user. However, the temporal fluctuation of geo-referenced streaming loads calls for an interactive controller based on the control theory and an appropriate cost model, that is able to respond adaptively to oscillations in data arriving paces and fluctuations in skewness. This has encouraged us to design an adaptive system that exploits SAOS in a control feedback mechanism, which, despite designed for heavy workloads (e.g., stream-static join), can be used for any streaming workload. We term the complementing sub-system as SpatialSSJP which is the topic of chapter 6.

Chapter 6

SpatialSSJP: Adaptive Stream-Static Spatial Join Processing

In today's big data landscape, online applications are, more than ever before, relying on real-time data for deeper insights that benefit businesses. IoT is currently the main source of huge avalanches of geo-referenced data streams that feed online services. It is expected that it generates more than 500 zettabytes of data by 2020 [107], which overburdens the capacities of current DSMSs. What’s more, most interesting application scenarios however contain mixed workloads requiring stationary data to be joined with data in-flight in order to pluck an interesting insight. The abundance of the ways we can mix data together led to the introduction of the Lambda architecture, designed specifically to handle low-latency updates in a linearly scalable way. Consisting of three layers, where streaming data is fed into either a batch or speed layer. Data can then be combined from both layers to be served through a serving layer to the benefit of dynamic application scenarios.

It is then becoming obvious that the join operation constitutes an integral building block of any successful SPE (a.k.a. Distributed Stream Processing System, DSPS for short). It is however prohibitively computationally expensive in an exhaustive way to compute over a huge amounts of fast arriving data streams and may take several hours to complete for complex querying scenarios [108, 109]. A problem that is exacerbated in geofencing that includes complex polygons [110].

As a natural resolution, *Approximate Query Processing* (AQP) (especially for ad-hoc and long-running queries) excels in optimizing the QoS of online join processing in highly dynamic and scalable application scenarios, such as those in smart cities [3] and Industry 4.0 [4]. AQP depends on the observation that an approximate answer that falls within the boundaries of a confidence interval suffices for expressing a statistical parameter. To address this problem, sampling-before-join seems a super-quick compelling fix, but, however, is a candidate for high accuracy loss as it may potentially deteriorates the statistical properties of study variables, simply because sampling designs commonly embrace randomness by depending on uniformity. On the other hand, sampling-after-join could be computationally prohibitive. Stated another way, in smart city scenarios, samples taken should be spatially

representative for the join after to perform well in the currency of estimating interesting geo-statistics. Approximate join methods from the relevant literature such as RippleJoin [111] and WanderJoin [112] are designed to be operating in single-node beefed-up servers and parallelizing them is challenging.

The fluctuating nature of arriving rates of data streams challenges the current Spatial Approximate Query Processing (SAQP) engines. It is hard to predict such an oscillating nature in streaming settings in addition to the temporal oscillating skewness. Such settings require an adaptable model-based solution that responds interactively to spikes. A compelling solution should also be able to control the way of join processing such as to achieve the prescribed SLAs, including most importantly the latency/throughput/accuracy QoS goals trade-offs.

To cope with the fluctuating rates of arrival data, another major goal of current systems, typically deployed over on-demand cloud environments, is to maximize resource utilization in order to lower the costs for the user. Accordingly, most state-of-art solutions depend heavily on elasticity, by overprovisioning and de-provisioning computing resources to maximize resource utilization. Nevertheless, studies have shown that the average utilization in cloud deployments is under 40 percent of the overall reserved resources [113, 114]. This is possibly due to the fact that users lack the relevant understanding on how to configure the auto-scaling parameters (which requires technical knowledge for most SPEs) that, in its turn, behooves them to select lenient configurations that allow, most often, the overprovisioning in order to handle peak loads , leading then to a low resource utilization. Consequently, elasticity methods, despite relevant for some cases, do not interplay well in resource-constrained settings. We consider systems with shortage in computing resources, being deployed on an on-demand cloud, where the goal is saving money by maximizing the utilization of resources, or deployed on in-house clusters, where the goal is to free unused resources for the benefit of other queued applications.

Our main goal in designing SpatialSSJP is to provide a QoS-aware optimization for online join processing in highly dynamic application scenarios. To realize this goal, we have designed an adaptive QoS- and spatial-aware system (we term as Spatial Stream-Static Join Processor, SpatialSSJP for short) for stream-static online join processing (i.e., joining

arriving tuples in a data stream with a static disk-resident master table). Most importantly, SpatialSSJP constitutes a controller, which contains two sub-controllers; the first is based on the control loop feedback mechanism and specifically the Proportional-Integrative-Derivative (PID) controller, which is utilized in case that the user prioritizes low-latency QoS goal. However, when the user chooses to prioritize high-accuracy (i.e., high estimation quality), then we apply the second model-based controller (described shortly in section 6.3.2.2) that returns results with rigorous error-bounds. The controller is entwined with our spatial-aware sampling method (Spatial Aware Online Sampling, SAOS for short) [101] that we have designed with SpatialSPE (refer to section 5.3.4 for details).

By lending SAOS, we guarantee that an appropriate count of items is safely purged from the arriving stream before applying the join predicate. A special characteristic of SpatialSSJP is that it preserves the spatial characteristics of the data stream. It is therefore aware of the data nature, and thereby preserving the geo-statistical properties of the served result by providing a spatial approximate result with rigorous error-bounds. Also, it meets the target QoS requirements prespecified by the user through SLAs from which two are most common and contradicting, high-accuracy (i.e., high estimation quality) and low-latency, where overoptimizing any may deteriorate the other in an endless tension. SpatialSSJP self-tunes the sampling parameter by calculating after each loop (i.e., batch or trigger in SPE terms) an appropriate sampling fraction (our solo configuration parameter, not inducing any extra overhead). Thereafter, it serves the sampling fraction back to SAOS module so as to select an appropriate spatially representative sample for the next batch (a.k.a. trigger in DSMS's jargon). All in all, satisfying the quality requirements prespecified by the user.

We have implemented SpatialSSJP on top of an emerging de facto standard general-purpose SPE, Spark Structured Streaming (SpSS hereafter for short) and evaluated its ability to achieve QoS goals by applying the general methodology that we have defined in section 3.2.1. We use a real-world data load against Spark's baselines (such as our glue for SpSS supporting backpressure mechanism) and the vanilla Spark without sampling. Our experiments show that SpatialSSJP is able to meet QoS goals prespecified by an expert user. In addition, it outperforms baselines in terms of accuracy and latency QoS attributes.

To sum up, we make the following contributions by designing SpatialSSJP:

- We develop model-based adaptive hybrid (reactive and proactive) controller for spatial stream-static join operators (i.e., equivalent to geofencing [110]) and incorporate it with an emerging SPE, specifically, SpSS, taking full advantage of the optimizations provided by the underlying layers of SpSS codebase.
- We enrich the controller by an accuracy-aware module (reactive) that receives a '*margin of error'* as an estimation-quality QoS goal and responds accordingly.
- We incrementalize the appropriate spatial statistics so that the performance improves as time ticks forward. We also support the incrementalization of other basic standard spatial queries such as Top-N ensembles and other spatial online aggregations.

To the best of our knowledge, we are not aware of any system from the relevant literature that synergistically achieves these goals. We first introduce the theory behind our work in § 6.1. We then explain the architecture of SpatialSSJP in § 6.2 in addition to a usage model and a baseline system. In § 6.3 we expand the algorithmic perspectives within the layers of SpatialSSJP, showing also the supported queries and measures to quantify the uncertainty. In what follows, we shortly recapitulate the implementation of SpatialSSJP in § 6.4. Thereafter, we present our results with proper discussions in § 6.5. This is followed by a short review of the relevant literature in § 6.6. We finalize by summing up the effort and recommend future research directions in § 6.7.

6.1 Background

We aim by this short background at assisting, in a coherent and structured way, to grasp the rudiments of the ideas presented in this chapter.

6.1.1 The Problem of Poor Resource Utilization in Stream Processing

The parallelized versions of SPEs distribute DAG operator instances to multiple worker nodes (i.e., bare-metal or virtualized) to achieve the primary QoS goals. But this sometimes cause the overprovisioning of resources thus lowering the resource utilization, which counteracts the benefits of parallelization. SPEs then should aim at maximizing the resource utilization in parallel distributed settings. For example, by releasing resources in in-house

computing clusters and make them available for other applications. Also, minimizing the cost incurred by the pay-as-you-go model while deploying on a Cloud infrastructure, but at the same time keeping the latency low. In summary, a goal not-to-be-underestimated is keeping the average end-to-end latency bounds low while maximizing resource utilization. Two highly contradicting QoS goals that can be exacerbated in continuous queries that contain complex operators such as the join operator. Coming up next is a subsection that summarizes the complexities associated with DAGs that encompass a join operator.

6.1.2 Streaming Distributed Joins and Complexities Associated with Spatial Cases

Data aggregation remains one of the most desired analytics in real-time applications. It heavily depends on joining data between either several streams (i.e., stream-stream join) or a stream and a static table (i.e., stream-static join).

However, in streaming scenarios, where data tuples arrive in an unbounded fashion that exhibits temporal skewness and fluctuation, results are normally incrementalized in unbounded manner. Hence, the assumption that input data is indexed does not play well with those settings, rendering standard join algorithms unsuitable in such streaming scenarios [115].

The parallel distributed processing model encompasses logistic complexities that are uncommon in traditional centralized single-server settings. For example, distributed processing engines (such as Apache Spark [1]) dispatch data loads to parallelly connected computing nodes aiming at speeding up the processing phase. Some operations however require shuffling data between nodes. Join processing is a potential candidate as joins can only be performed on same-node basis. SPEs normally apply either *repartition* or *broadcast* joins (refer to section for 2.3.1 details). In cases where a static master table is small enough to reside in-memory of all processing worker nodes, it is broadcasted (together with a join operator instance) and the join that is then performed needs no shuffling as it is done locally, from which conquering local join results is the only thing at the time that remains incumbent (usually is achieved by the master node). However, if the master table (i.e., disk-resident) is huge, and therefore cannot fit in-memory (a.k.a. in fast memory), then data need to be shuffled around in the so-called repartition join, which is computationally expensive and resource-hungry. Cases where joins take several hours or days are not unheard of, especially

when exposed to big data on the scale of terabytes to zettabytes. For thorough details of the mechanics of those joins in Spark, we refer the interested reader to [116] . From many join types that are supported by SPEs, we specifically focus on a class that is most widely used in highly dynamic and scalable scenarios. That is the so-called stream-static join (a.k.a. semi-stream join [117] or Stream-Relation Join (SRJ) [118]) which aims at joining on-the-fly online arriving tuples with a master static table data (i.e., disk-resident opponent). Join is natively computationally expensive in its simplest forms. A problem that is further inflated when applied in specific dynamic application scenarios such as smart cities and Industry 4.0. This in part is because those data tuples are geo-referenced (for example, in the form of longitude/latitude coordinates). Joining spatial data streams costs more. Take the case of Point in Polygon (PIP) test which necessitates the application of the costly '*within'* spatial join predicate, as an example. A typical query could ask to "find in which borough (i.e., polygon) of NY city (in the United States) a taxi trip (i.e., spatial point) has started". This is specifically complex and challenging because of the "curse of multi-dimensionality", which means joining a GPS coordinate streaming tuple with a static table containing the covering polygons. In this case, simple traditional join algorithms (such as sort-merge join) are inapplicable, because in spatial joins, join condition comprise multidimensional attributes [119].

Stream-static join is of a paramount importance also in other domains such as data lakes and active data warehouses [120] . For example, in data warehouses it is important for *surrogate key* generation, duplicate detection or identification of newly inserted tuples in view maintenance scenarios.

Stream-static join constitutes thus a strongly potential candidate for optimization through controllers as explained in the next subsection. The rationale is that distributed stream processing (e.g., stream-static join) runs into multiple complications that do not normally affect simpler computations like batch jobs (e.g., static-static join). For example, peak loads that exceed resource computational capacities.

6.1.3 Controllers for Resolving the Information Overloading and Resource Utilization

Information overloading is presenting current big data management systems with tremendous challenges and obstacles. Online streaming data shows temporal skewness and fluctuation, thus challenging, at times, the capacities of existing parallel computing deployments. In cases where a peak load emits excessive amounts of data that outpaces the processing capacities of the operators, data accumulates excruciatingly upstream, causing congestion in the operator input queue that carries over a negative effect, deteriorating the online processing. Also, most cloud deployments underuse provisioned resources, thus minimizing the resource utilization. The unbounded fashion at which streaming continuous queries work requires the innovation of adaptive mechanisms that can survive brutal burst workloads at peak times. However, most traditional methods depend on a presumption that data loads are finite, rendering them inappropriate for unbounded semantics [115].

Many solutions have been widely used in the literature for resolving the information overloading in a manner that maximizes resource utilization. Most of them depends on elasticity and adaptivity. We identify three most widely used alternatives. Those are backpressure, elasticity and approximate computing. We describe each one in details in the following subsections.

6.1.3.1 Backpressure for Resolving Data Load Bursts

Situations where streaming data arrival rates (e.g., during a temporary load spike) exceed the capacity of the receiving processing engine are not unheard of, which can cause bottlenecks in downstream dependencies. Backpressure has been widely used as a solution, which can be loosely defined as a mechanism that pushes back the lateness to the ingestion layer (such as Apache Kafka [43]) by only allowing the sender to emit an acceptable rate of tuples that can be processed gracefully without causing batches to back up.

Backpressure normally depends on a rate limiter in the back-end, which is then responsible for calculating, at each batch interval (trigger in SpSS jargon), a new rate that the system capacity can handle without falling behind. This ensures that the system is stable (i.e., scheduling and processing delays are not stacked up).

Streaming systems often depend on feedback loop mechanisms such as PID algorithm to build the rate limiters. A case of example is Apache Spark Streaming [25]. *PIDRateEstimator* in Spark Streaming first calculates a rate at which its receiver (a building block in Spark Streaming) is writing data blocks to a block manager (a building block in Spark Streaming). If the received data is outstripping the processing capacity, then the new rate is decreased, while it will be increased in the opposite case.

However, backpressure has been abandoned in many systems because of the negative effect in processing fast streams. Among others, it might congest data receivers, which wait for the overloaded downstream operators to finish processing, which normally leads to endanger QoS goals.

6.1.3.2 Elasticity and Adaptivity: to Assign or to Release?

The ability of the system to elastically change the parallelism degree (mostly at run time) based on the trending circumstances (i.e., peak data loads as opposed to off-peak loads) to achieve the QoS goals is known as elasticity [121]. With the introduction of the pay-as-you-go models (e.g., in cloud deployments), SPEs are currently seeking to strike a balance between provisioning extra resources (which is costly but strong against oscillating data arrival rates) and resource utilization (which is cheaper but vulnerable to unpredicted data load spikes). By this strategy, they aim at maximizing the latter while keeping the former at the bare minimal level in order to cut costs associated with overprovisioned nonutilized resources. An obvious problem is that the continuous process of provisioning and de-provisioning resources dynamically may counteract the benefits of elasticity, especially in cases where the costs associated with always-on reconfigurations are not amortized by the benefits of elasticity (e.g., reducing latency).

This kind of resource scaling is sometimes referred as *dynamic allocation*, including horizontal scaling (a.k.a. in/out) on clusters of commodity servers, where extra computing nodes (or virtual machines) are added to a cluster of computing resources connected parallelly. This can be done dynamically online as a resort for sudden spikes in data arrival rates (dynamic allocation in Spark Streaming parlance [25]). Another type of dynamic allocation is the vertical scaling (a.k.a. up/down), which means adding extra computing power (e.g., CPUs and memory) to a single node, normally a beefed-up server. Auto-scaling

techniques [122] depend on diverse approaches for the decision on when to scale, including threshold-based and control-theory based.

In summary, the near-linear scalability provided through provisioning extra computing resources in cloud deployments is no longer attractive as it comes on the cost of inefficient resource utilization and deteriorated throughput.

6.1.3.3 Approximate Computing

All approaches discussed thus far focus on the assumption that there are readily available on-demand resources to allocate to a join operator dynamically on-need. However, the case where only a fixed memory is allocated, where data load spikes exceed the operator service rate, are interesting. In those cases, there is a point where the operator cannot withstand the transient burst load that easily turns insurmountable at times, a point that requires employing an approximate computing.

Approximate computing depends only on a portion of input data to get results in what so-called Approximate Query Processing (AQP), which means basically serving results that are bounded by rigorous error-bounds in a form that is statistically acceptable and plausible. It is based on the observation that users are accepting normally to tradeoff tiny accuracy for a high speedup [123]. Also, decision makers normally make perfectly accurate decisions without having perfectly accurate query responses (for example, the cases of A/B testing and visualizations such as in heatmaps). In addition, the data is normally noisy and depending on a whole population in query answering does not readily imply accurate answers. A special branch of AQP is the so-called Spatial AQP (SAQP), which is then the same as AQP, but becoming more attuned to the shape and structure of the data (i.e., spatial data in this case). We can reap tremendous benefits by being attuned of the spatial structure of the data stream. AQP depends on shrinking the input data size based on diverse mechanisms, including sampling and backpressure. Approximate query processing via sampling is a popular technique. We opt for approximation in latency-sensitive settings over resource-based elasticity approaches because of two main reasons.

1) AQP does not imply that system halts processing during adaptation. In elasticity approaches, on the contrary, each reconfiguration halts data processing, and thus negatively affects quality of service (e.g., end-to-end

latency) [124]. In our system, the reconfiguration cost is tiny and negligible as we only change the sampling fraction that is passed to SAOS. Other information for computing the fraction provided for us already by SpSS at no additional cost.

2) The management of state migration (online migration in this case) in elasticity approaches is costly, which may trigger at the time of altering the parallelization degree of a stateful operator, which may require , for example, re-splitting the keys, and thereby broadcasting the key state again. In most current SPEs, this means halting the processing until state migration is done, which then incurs extra latencies.

6.1.3.4 QoS-aware Sampling as an Enabling Technique for Spatial Approximate Computing

Recap sampling types and SAOS from chapter 5. Sampling as a mixed workload with join can be performed either before or after join. If performed before join, sampling focuses on selecting a sample and then applying the join on it (sample is taken from the stream in the stream-static scenarios). We focus on sampling before the join because the main goal of this work is to make the join adaptive and limiting the arrival rate to quantities that do not exceed DAGs capacities.

Despite the abundance of alternative AQP methods such as sketches and wavelets, we found that sampling is the most compelling and a powerhouse method to be used in SAQP because of the additive property. In other terms, taking a simple sample, computing the incremental result and if the result is not satisfying the QoS stringent goals, then adding incrementally more data to the sample does not mean recomputing or reconfiguring, instead the result is building up on the previously obtained sub-results and the incremental procedure can be continued indefinitely until either a predefined rigorous error-bound or a latency goal is achieved. On the contrary, for other AQP methods, this does not apply, and a result that is not satisfying could require a recourse that involves recomputing using the whole sketch (i.e., synopsis).

6.2 QoS- and Spatial-Aware Adaptive Stream-Static Join Processor

In this section, we describe all the peculiarities associated with the SpatialSSJP system that we have designed, which aims basically to proactively avoid congestion within the operator graph (that specifically includes a stream-static join operator), and thus seeking high resource utilization and averting frequent reconfigurations.

6.2.1 Usage Model and Baseline System

While the main purpose of parallelizing the operation of SPEs is to achieve low latency and high throughput, there are innumerable scenarios that require accessing static information (i.e., information that is held and spelled out in disks), thus compromising the performance of the SPEs [118] . There are innumerable ways for which stream-static join is attractive. Always focusing on highly dynamic and scalable scenarios, where fast arriving spatially-tagged data points need to be enriched with master static data (a.k.a. data-at-rest) for deeper insights.

For example, NYC taxicab trips (represented with, most importantly, pickup and drop-off points) have been distributed on the form that includes only the GPS longitude/latitude coordinates without the names of the regions that those traces belong to. On the other hand, names of zones (i.e., boroughs or districts in city management terms) are distributed alone in a static table. That table is normally containing polygons on the form of points covering each polygon (a.k.a. bounding box). An example scenario is a query that asks to “generate an interactive heatmap showing trajectories of taxis in-motion to see the trend and decide on city planning”. As such, specifying the neighborhood for every tuple requires solving the Point in Polygon (PIP) problem (a.k.a. geofencing [125]), which basically requires stream-static join. As the amount of streaming data can be prohibitively large in the terms that our screens are not able to efficiently absorb such amounts in one map (e.g., while generating heatmaps), then it is favorable to take only portion of the arrival data and join it with the static table.

However, the tremendous deluge of geo-referenced continuously arriving data streams challenges the capacities of currents SPEs in achieving a (near) real-time interactive visualization (e.g., through heatmaps). A spatial-aware online sampling is then necessary for a proper data reduction, thus striking an acceptable balance between accuracy and latency

QoS targets. This reduction requires either clustering or aggregating, which basically, in a streaming setting, means joining tuples (that are spatially-tagged) with data-at-rest, thus the stream-static join. Traditional sampling designs (such as SRS, refer to section 5.2.2 for information) do not consider the spatial characteristics of the arriving tuples, thus rendering the visualization process erroneous.

Approximation is a valuable solution in highly dynamic environments. Baselines include a standard-compliant system employing backpressure on top of an emerging de facto standard SPE, specifically SpSS. The SpSS baseline is a resemblance to that of Spark Streaming backpressure mechanism. Spark Streaming backpressure works by applying a PID controller (known as PID rate estimator in Spark Streaming, which is based on the PID theory). We have retrofitted the PID controller (similar to *PID rate estimator* in Spark Streaming) so that it transparently incorporates with SpSS and operates under the SQL-like API. The baseline also comprises SRS-based sampler instead of SAOS. That is for the case of low-latency QoS goal.

As a baseline to compare our models for the case of accuracy QoS goals, we have transparently incorporated within the layers of SpSS a model-based controller that is based on SRS theory for calculating a new sampling fraction after every trigger and serving it interactively to an SRS-based sampler (as opposed to our SAOS sampler).

6.2.2 SpatialSSJP Overview

We have designed SpatialSSJP[12] (short for Spatial-aware approximate Join Processor), an adaptive QoS- and Spatial-aware framework for processing spatial stream-static joins efficiently. Our system employs hybrid model-based controllers to reactively and proactively handle the information overflooding during burst spikes in spatial data streaming workloads.

[12] The source code of SpatialSPE (including rateController) is available at: https://github.com/IsamAljawarneh/SpatialSSJP

We depend on hybridizing a novel rate controller with our spatial-aware sampling method (from our previous work SAOS [101] , refer to section 5.3.4 for details). A general overview of SpatialSSJP is schematized in the context diagram of figure 6.1.

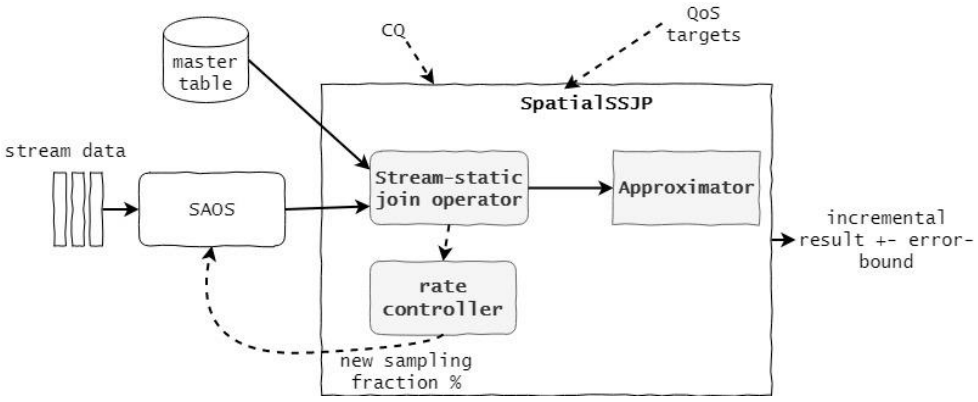

Figure 6.1. SpatialSSJP Overview. CQ is 'continuous query'

An expert expresses the continuous spatial query (that implicitly requires stream-static join operation) and a query running budget. Budgets are a form of either latency or accuracy QoS guarantees. Data is arriving continuously from geo-referenced heterogeneous sources and is then fed interactively at regular time intervals (e.g., batch intervals, a.k.a. trigger intervals in SpSS terms). We have implanted our cogent method SAOS in a front stage so that it receives a signal from the rate controller of SpatialSSJP that informs the new appropriate sampling fraction. SAOS then selects a proportionate sample and emits it to the stream-static join operator, which thereafter forwards the intermediate result to the approximator. Approximator completes the approximate computation cycle and serves an incremental result with rigorous error-bounds to the user. At the same time the join operator sends statistics of the latest trigger to the rate controller, which exploits those statistics in calculating new sampling rate and serving it to SAOS to be applied in the next trigger. SpatialSSJP comprises three main components:

- **Stream-static join operator**. This component is responsible for stream-static join over the sampled subset. While stream-static join processor in SpSS is a simple and conceptually appealing approach, it suffers from computational limitations when

applied to spatially-tagged datasets. Our join operator is alternatively then a retrofitted version based on an operator offered by the spatial-aware library on top of spark (known as Spark's Magellan[13] [12, 13]). Spark's Magellan is basically designed to handle static-static spatial joins using z-order curves. We have retrofitted Spark's Magellan so that it works with the stream-static join, using the primitive features of Spark's Magellan that offer basically a static-static join.

- **Rate controller**. The rate controller depends on QoS goals fed to the system by an expert user. Our controller is composed of two sub-components; *latency-aware rate controller* and *accuracy-aware rate controller*. For latency-aware rate controller, we have incorporated a hybrid (i.e., proactive and reactive) model-based loop feedback mechanism for appropriately pruning the arrived data loads to avoid system failure and achieve the latency QoS targets. The controller calculates a new appropriate rate (which is then mapped into an appropriate sampling fraction) and feeds it back to the SAOS method to force SAOS to limit the rate of data accepted for processing in the next trigger. *Accuracy-aware rate controller* employs a model-based statistical approach to compute a sampling fraction that is appropriate for meeting the accuracy requirement (expressed as '*margin of error'* value, explained shortly in section 6.3.2). Notice that both controllers have one common reconfigurable parameter, which is the *sampling fraction*. Since sampling module is a front-stage, then the overhead caused by the rate controllers is tiny and negligible, which is highly desirable in the control theory.
- **Approximator.** This component is responsible for receiving the output of the join operator and then using the result in incrementalizing a required statistical target variable. For example, calculating the "average trip distance for all Uber trips (e.g., during a specific time-based window) originated in a specific district in Amman city (in Jordan)".

[13] https://github.com/harsha2010/magellan

6.3 SpatialSSJP Algorithms and Mathematical Formulations

In this section, we explain the algorithm that shows the workflow of SpatialSSJP in addition to the algorithms of the rate controllers.

6.3.1 SpatialSSJP Workflow

SpatialSSJP workflow is given in Algorithm 6.1, including the procedure for implementing stream-static join based on a retrofitted version of Spark's Magellan-based spatial static-static join.

Algorithm 6.1. SpatialSSJP Workflow

/ latThrAccTargets: latency throughput and accuracy targets*

*geoPrec: geohash precision */*

Input: stream, ContinuousQuery (CQ), latThrAccTargets, polygons, geoPrec

coverGeo ← getCoverGeo (polygons, geoPrec) */* List of geohashes covering each polygon */*

//cost model computes the sampling fraction
newSampFraction = 1.0 *//initially do not sample*
While true *//loop forever – unbounded stream*
Foreach time window interval **do**
  windowSample = ∅ *// tuples sampled in current time window*
  Foreach batchInterval **in** window interval **do**
    batchSample = ∅ *//tuples sampled in current batch interval*
    Forall $tuples_i$ **in** batchInterval **do**
      / apply SAOS on tuples of current batch interval: $tuples_i$ */*
      batchSample ← SAOS ($tuples_i$, samplingMap, NewSampFraction, seed)
      windowSample.add(batchSample)
    End
  End
/ perform inner join on geohash using the filter stage, filter-and-refine approach */*
joinResult = windowSample.join(coverGeo, windowSample("index") == coverGeo("index"))
/ refinement stage, filter-and-refine approach, by applying the 'within' join , refer to listing 6.1 for an example */*
optimizedJoinResult = joinResult.filter(edgeCases)
  //Compute and serve incremental output every time window
newSampFraction ← rateController(*latThrAccTargets*) // *section 6.3.2*
incrementalOutput ← run (CQ, optimizedJoinResult)
return incrementalOutput with error-bounds
End
End While

Listing 6.1 shows an example stream-static join processing using a retrofitted version of Spark's Magellan. Refer to section 2.3 for more details.

```
pointsDF.join(polygonsDF,pointsDF("index")==
polygonsDF("index")).where($"point" within $"polygon")
```

listing 6.1. An example stream-static join processing using Spark's Magellan

To sum up, the three constituent building blocks of SpatialSSJP are: (i) stream-static join operator, (ii) rate controller and (iii) approximator. Those are interweaved and bounded together in a synergistically complementary way so that the benefits accrued by their synergy is greater than their combined independent benefits. Data passes through SAOS (initially disabled in the first trigger, batch interval) to be then fed to the *stream-static join operator* that performs the spatial-aware join (through a retrofitted version based on Spark's Magellan) and the join results are forwarded to the *approximator* that computes the CQ . During the CQ computation in every trigger, *rate controller* module computes the new sampling fraction based on the query budget and send it back to the SAOS module to select a proportional sample. After each window interval results are served to the user, either achieving the latency target (currently stepwise, the mechanism in which PID works) or the geo-statistically plausible rigorous error-bounds (e.g., in the form of confidence intervals).

6.3.2 Rate Controller Algorithm

The procedure rateController workflow is given in Algorithm 6.2.

Algorithm 6.2 rateController Procedure

```
1:  Procedure rateController (latThrAccTargets)
2:      If (priority == latency)
3:          rate_new = LatencyAwareController(latencyTarget, PIDvalues)
4:      Elseif (priority == accuracy)
5:          rate_new = AccuracyAwareController(marginOfError)
        End if
6:      Return rate_new
7:  End procedure
8:  Procedure LatencyAwareController(latencyTarget, PIDvalues)
    /* retrieving statistical information from the last trigger,
    Processing time, and number of elements */
9:      lastTriggerInformation = retrieveLastTriggerInfo( )
```

/* *adapted from Spark Streaming* [22, 126] *but here applied to*
13: $rate_{new} = rate_{latest} - (p.err) - (I.err_{hist}) - (D.err_d)$
14: **End** Procedure
15: Procedure AccuracyAwareController (marginOfError e)
16: $rate_{new} = z_{\alpha/2}^2 v / {e_{des}}^2$
17: $rate_{new} = 3.84 * (v/{e_{des}}^2)$ // $v = \sum_{h=1}^{H} {}^{n}/_{n_h} (N_h/N)^2 S_h^2$
18: **End** Procedure

We offer a simple interface that allows an expert to express targets as either desired latency or the desired rigorous error-bound ($\boldsymbol{e_{des}}$). Currently, PID controller eliminates the latency stepwise. Our rate controller then guarantees that the stream-static join is performed within the budget. It does so by calculating an appropriate sampling fraction depending on one of two procedures as explained in the next two subsections.

6.3.2.1 Latency-aware Rate Controller

SpatialSSJP takes latency QoS goals specified in the query budget and then applies a retrofitted version of PID controller. PID controller is a control loop feedback mechanism that calculates an error value by subtracting a measured process variable (i.e., PV) from a desired setpoint (i.e., SP). PID controller then enforces a correction depending on three terms known as *proportional*, *integral*, and *derivative*. The process aims to settle the PV by reducing three error values. In our setting (and similarly those of Spark Streaming backpressure version [22, 126]), *proportional* term defines how correction depends on the present error (w.r.t. the latest measurement from the latest batch information). *Integral* term specifies the way that the correction should react to the accumulation of historical errors (i.e., accumulated through past triggers or 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 depend on future errors prediction based on error change between two triggers (i.e., the trend).

As the time of this writing, backpressure through PID controller has not been applied to Spark Structured Streaming or incorporated with a sampler for dropping loads in a convenient way that achieves high incrementalized *estimation quality*. To close this void, we have retrofitted the plain version of the PID rate controller (known as PID rate controller in

Spark Streaming terms) that has been applied in Spark Streaming [22] for backpressure [126] so that it transparently incorporates within the layers of SpSS. It worth mentioning though that for calculating the three terms of PID (i.e., Proportional, Integrative and Derivative), we use the same mathematical model-based formulation approach as the one that was applied in the plain Spark Streaming version. After each trigger, the new rate is calculated with (6.1).

$$rate_{new} = rate_{latest} - ((P.err) + (I.err_{hist}) + (D.err_{d})) \quad (6.1)$$

Refer to Appendix F for a detailed explanation on how each term is calculated with equations adapted from the PID application in Spark Streaming [22].

It worth mentioning though that PID controller has been used in a similar way by Spark Streaming with the same formulation to activate the backpressure mechanism. But however, it has never been used to activate a SAQP with the declarative API (i.e., SQL-alike API in SpSS), especially in a spatially-rich environment. To the best of our knowledge, we are not aware of any system in the relevant literature that achieves these goals.

In this thesis, we do not focus too much on future prediction. The rationale is that the relevant literature proved that only adaptive approaches that place no (or at most very little) assumptions on workload characteristics are considered stable and may show good performance for data stream processing systems, since workloads oscillate continuously in unpredictable ways [127] .

6.3.2.2 Accuracy-Aware Rate Controller

If among SLAs there is a '*margin of error'* specified as a QoS target, then our rate controller activates the mode that computes a new sampling rate based on the error-bound specified. Since our SAOS method resorts to stratified sampling in its core, then we depend on the theory of stratification [90] for estimating a proper sample size depending on a prespecified '*margin of error'*. As such, some equations in this section are adapted from [90]. We specifically depend on (6.2).

$$n = z_{\alpha/2}^2 v / {e_{des}}^2 \quad (6.2)$$

Where we calculate v depending on (6.3).

$$v = \sum_{k=1}^{K} (n.{N_k}^2.S_k^2/n_k.N^2) \quad (6.3)$$

This approach supposes that we have some knowledge of v perhaps from a previous survey. As this may not potentially be the case in streaming settings, we otherwise depend on incrementalization and loop feedback mechanism in improving the value of v after each trigger and feeding it back to the controller. Other possible approaches include profiling some data in a method dubbed as bootstrapping [90]. We however avoid that approach in this thesis. The reason is that we target settings where profiling is not easily accessible. With 95% confidence level, we have $z_{\alpha/2}$= 1.96; thus, we apply (6.4) to calculate the new rate.

$$n = 3.84 * (v/{e_{des}}^2) \quad (6.4)$$

Which then calculates an appropriate sample size given a 'margin of error e'. For a fair comparison, as we are comparing the employment of SAOS in the front-stage against and SRS-based design, we also depend on the theory of simple random sampling [90] for estimating an appropriate sample size based on a target '*margin of error'* in cases that SRS is applied instead of SAOS. We specifically employ (6.5),

$$n = (n_0.N)/(N + n_0)) = 1/(1/n_0 + 1/N) \quad (6.5)$$

to calculate the desired sample size. If the population size N is large relative to the sample size n so that the finite-population correction (fpc) factor can be ignored, the formula for sample size simplifies to $n = n_0$. Where n_0 is calculated using (6.6).

$$n_0 = z^2\sigma^2/{e_{des}}^2 \quad (6.6)$$

It is then apparent by combining the two equations that we obtain a tradeoff equation between latency (incorporated within the three terms of PID) and '*margin of error'* (i.e., accuracy, *estimation quality*) which is shown in (6.7), rendering the problem a conundrum where optimizations are limited.

$$rate_{new} = rate_{latest} - ((p.err) + (I.err_{hist}) + (D.err_d)) = 3.84 * (v/e^2) \quad (6.7)$$

6.3.3 Supported Queries

We support online spatial aggregation (was first proposed by [128]), where join is part of the query plan. Hence, we are interested in an end-to-end accuracy (i.e. estimation quality), as it is hard to factor the join operator independently. 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 [129]. Some equations in this section are adapted from [90].

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 $P(|\bar{y}_{samp} - \bar{y}_{pop}| \leq \boldsymbol{e_{des}}) = 1 - \alpha$, where $\bar{y}_{samp}$ is the estimate of the 'mean' value using the sample, $\bar{y}_{pop}$ is the estimate of the mean using the population, and e_{des} is the permitted error (i.e., *margin of error)*. The investigator normally decides acceptable value for α and e_{des}. For example, e_{des} = 0.02 and α = 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' 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.

6.3.4 Quantifying Uncertainty

We depend on the same set of equations of chapter 5 (specifically, section 5.3.6) for quantifying the uncertainty carried by the estimations through sampling instead of the population. In addition to the following new equations. Some equations in this section are adapted from [90].

We first depend on an equation that is adapted from [130] to certify that samples drawn are always falling with the minimum standard recommended for the normal approximation to be adequate, which is given by (6.8).

$$n_{minimum} = 28 + 25\,(pop_{skewness})^2 \tag{6.8}$$

, where $pop_{skewness}$ is the population skewness that is calculated using $\sum_{k=1}^{N}(y_k - \bar{y})^3 \Big/ (NS^3)$ adopted from [90] , which is then responsible for specifying the

size of the sample for the normal approximation to be accepted, where large skewness signifies the need for a large sample size and vice versa.

For single queries, in addition to those in section 5.3.6, we also rely on '*coefficient of variation'* **(**CV) [90] as a measure of relative variability using (6.9).

$$\widehat{CV} = \frac{SE(\bar{Y}_{SAOS})}{\bar{Y}_{SAOS}} \quad 6.9)$$

Which is then equivalent to the SE as a percentage of the mean. In addition to that, we calculate the gain of applying SAOS (instead of the SRS-based baseline), for which we use the design effect (abbreviated *deff*) [90] , which provides a measure of the precision gained or lost by using a more complicated design instead of an SRS. *deff* is computed using (6.10).

$$deff = gain_{SAOS} = \frac{estimated\ variance\ from\ SAOS}{estimated\ variance\ from\ SpSS-based\ SRS} \quad 6.10)$$

6.4 Implementation

To implement SpatialSSJP, we have built a standards compliant prototype on top of the elastic data SPE Spark (specifically SpSS). Also, because by our work presented in this thesis, we aim at complementing an end-to-end QoS-aware pipeline for big data management in dynamic application scenarios, we aim at incorporating the work with our modular architecture that can achieve mixed workloads (recall SpatialDSMS from section 3.4). We have selected Apache Spark as a candidate to implement SpatialSSJP, and specifically we depend on SpSS [6] for the overarching traits that makes it excels in its class. Spark is the de facto best-of-breed standard for processing streaming mixed workloads. However, Spark in its vanilla version does not offer over-the-shelf spatial-aware services. A shortcoming that led to the emergence of spatial-aware glues and patches on top of Spark. We specifically depend on a spatial-aware static-static join library recently popularized (the so-called Spark's Magellan[14] [12, 13]) as it specifically employs a relatively fast dimensionality reduction

[14] https://github.com/harsha2010/magellan

approach (i.e., z-order curves imposed on a grid-based representation) in joining spatial datasets over Spark. Although faster than any state-of-art spatial-aware join methods, Spark's Magellan does not offer adaptivity and can collapse in spatial join scenarios where data arrives in tremendous amounts at spikes. However, Spark and its spatial library Magellan serve as good jumping-off points for a novel approach that is QoS-aware, which is our SpatialSSJP. One other reason that guided our decision in selecting Magellan and preferring it over counterparts is that it is the first spatial-aware library over Spark that extends Spark SQL [16] , thus inherently providing relational-alike abstractions for geospatial analytics (most importantly spatial join in this case). By doing so, SQL-alike queries are applied to DataFrames with geometric predicates (e.g. *within*, *contains* and *intersect*s).

Our approach is a top-down, where we start by tuning on top of a Spark Structured Streaming-based model (i.e., Spark's Magellan), which per se is internally tuning the catalyst model, and thus everything is compiled down to RDDs. Because of Spark modular architecture, we believe that this way we avoid reasoning about the underlying processes atomically (as recommended by the Spark development team [6]). We have implemented the system by coding basically using Scala on Spark.

Backpressure is provided off-the-shelf by Spark Streaming [25]. However, as the time of this writing, it was not incorporated with the SpSS [6] . For a fair comparison, and since we are layering up our architecture on top of SpSS, we have added a patch that implements and runs backpressure and glues it transparently within the layers of SpSS.

We have implemented the two rate controllers (i.e., latency-based and accuracy-based rate controllers) by adding our coding patches and incorporating them transparently within the layers of SpSS. First, for the latency-based controller, we have retrofitted the PID controller that has been used previously by Spark Streaming [25] for backpressure. We use the same mathematical model-based formulation from Spark Streaming. The novel contribution we provide is the incorporation of the PID controller within SpSS. Also, our version of the PID controller calculates a new sampling fraction after each trigger. On the contrary, the version implemented in Spark Streaming calculates only a new rate at which the DAG graph is able to process at any given moment and serves it to backpressure so as to limit the rate a data ingestion layer is emitting. For the accuracy-based controller, we have designed a model-

based controller that calculates statistics after every trigger and uses that information for calculating a new sampling fraction. The overhead induced by extracting that statistical information is negligible and mitigated by the QoS benefits we reap.

The user can express the query in SQL-alike format such as the example in listing 6.2. In this query the user wants to specifically "compute a statistical attribute (i.e., average) of a target variable (i.e., trip distance) and then aggregates (by neighborhood attribute) after performing the stream-static join". The join operation is performed by applying the *filter-and-refine* approach (recall information from section 2.3.1). By using geohash indexing (a class of z-order curves), we have reduced the join predicate in the *filter* stage from a spatial predicate (i.e., '*within'* predicate in this case) into a simple equal predicate (i.e., MBR-join). However, in the *refinement* stage, the costly '*within'* predicate still need to be applied to discard the edge cases (i.e., BSOs). The user expresses this continuous query with an incorporated query budget as shown in listing 6.2 and then serves it to SpatialSSJP.

```
SELECT point p, polygon po, avg(tripDistance)
FROM Stream S, MasterTable M
WHERE S.key = M.key AND (p WITHIN po)
GroupBy neighborhood
Latency 120 MS
OR
e 0.03 CL 95%
```

listing 6.2. An example spatial approximate online aggregation query with QoS goals

The stream-static join in this query will be compiled down into two parts. The first part is an equijoin (the 'S.key = M.key' operation in listing 6.2), which is analogous to the *filter* stage in the *filter-and-refine* approach. This part will be executed using the relatively cheap MBR-join (refer to section 2.3.1 for details). The second part (the operation 'p WITHIN po' in listing 6.2) requires applying the *refinement* stage from the *filter-and-refine* approach (recall section 2.3.1 for details), which then executes the costly join predicate (i.e., PIP test, 'within' predicate in this case). The purpose of executing this refinement stage is to eliminate the edge cases (i.e., BSOs). This is equivalent to the spatial query of listing 6.3.

```
SELECT point p, polygon po
FROM point SPATIAL JOIN polygons
WHERE WITHIN (p, po)
```

listing 6.3. an example of an exhaustive PIP test

SpatialSSJP then executes the query either within a “best effort” stepwise strategy to reduce the latency so as to approach the target latency specified (or even less) or it works at achieving the accuracy level (i.e., estimation quality QoS goal). All in all, we support the same set of queries that we previously supported in our previous work (SpatialSPE [101] , chapter 5). The addition here is that the join operation enables a coarser granularity. We measure then the accuracy of the queries with a coarser granularity. For example, aggregating on the neighborhood level (i.e., a coarser level) instead of the geohash level (i.e., fine grained-level, the level we natively supported in SpatialSPE [101] , as explained in chapter 5).

6.5 Performance Evaluation and Results

In this section, we discuss the deployment settings, data used for benchmarking, and how SpatialSSJP excels in meeting QoS targets specified though SLAs.

6.5.1 Deployment Settings, Test Cases and Benchmarking

Dataset. We use the same dataset cohort that we have exploited for SpatialSPE (refer to section 5.5.3 for details).

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 computing nodes in total (2 Head nodes, analogous to master nodes in Amazon, plus 4 worker nodes). Head nodes 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.

Testing scenarios. We have developed complicated mix workload scenarios that require stream-static join, we aim to measure the following.

1) *SpatialSSJP ability to satisfy a target latency requirement by applying the latency-aware rate controller*. We compare that between applying SAOS and SRS-based sampling approaches. For this scenario, we use two PID values settings. For the first, we use P=1, I=1, D=1. For the second, we use P=1, I=0.6, D=0.2. by alternating values of PID, we are able to measure the effect of the degree of the term consideration. For example, the less means that we give less importance to the associated term. For example, in the second setting we set D=0.2 to say that we do not want to affect the system stability by accounting for a future prediction too much. Instead, we consider future load trends slowly. We do the same in cases of using SAOS and SRS-based sampling. Also, we mimic the oscillating nature of arrival rates by fluctuating rates in diverse settings. '500K to 2000K', '500K to 3000K', '500K to 5000K', and '500K to 2000K to 1000K'. By doing so, we measure how the system responds to oscillation. We compare SpatialSSJP with backpressure (a patch that we have added atop SpSS) using the same settings. In addition, we alternate the geohash precision between 30 and 25, aiming at measuring the effect of granularity in latency gain and to see how fast the system can recover and be controlled back to a normal range.
2) *SpatialSSJP ability to satisfy accuracy target by applying the accuracy-aware rate controller*. We fix the arrival rate and change the accuracy target (i.e., margin of error) between 0.01 (strict and stringent), 0.03 (middle strictness) and 0.09 (permissive). We compare the join under SAOS against the join by using SRS-based sampler. Backpressure is not applicable in this case as one of the shortcomings that detract backpressure is its obvious inability in achieving a desired accuracy target in a timely fashion. The reason is that backpressure pushes the lateness upstream forestalling the emitters from sending new data until the operators downstream have a capacity.

6.5.2 Results and Discussion

All results reported in this chapter are calculated as the median (i.e., 50^{th} percentile) of ten runs.

6.5.2.1 SpatialSSJP ability to satisfy a target latency requirement

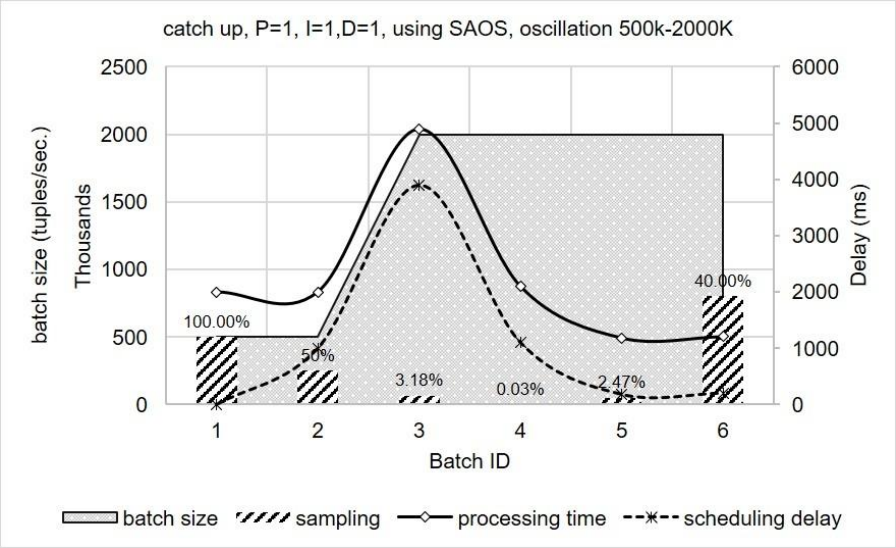

Figure 6.2. catch up at PID values 1,1,1 where SpatialSSJP is able to meet the latency target by applying the latency-aware controller. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

Figure 6.2 depicts how the latency-aware controller of SpatialSSJP is able to lower the latency to the minimum (near zero) which was specified as a latency target by the user. PID values used in this case are (1,1,1), respectively. Scheduling and processing delays (processing delay is equal to processing time minus the batch interval) converges at the first point where the latency-aware controller realizes that a staggered delay is caused by the sudden spike in the batch size (i.e., from 500K to 2000k) and a newly computed sampling fraction which roughly equals to 3.18% is then served to SAOS, after that a catch up occurs, but then the controller decides to take it slowly (because of the D value being 1, accounting more for a future possible sudden spike in the batch size), thereafter the system returns back to normal operation stepwise slowly increasing the fraction rate after each trigger (reaches on the verge of 40% at batch ID 6).

Trigger interval in all those experiments is 1 second (1000 milliseconds). Notice that processing delay (processing time) never goes below the duration of the trigger interval. This is because we depend on the tumbling window semantics (i.e., non-overlapping time-based

windows). Notice that using PID, the lateness is amortized stepwise and even if the flood slows down it returns to a previous state stepwise to account for future spikes.

For the same oscillation settings, but changing the PID values to 1,0.6,0.2 respectively (scientifically plausible margins), we obtain the adaptation shown in the visual representation of timeline in figure 6.3.

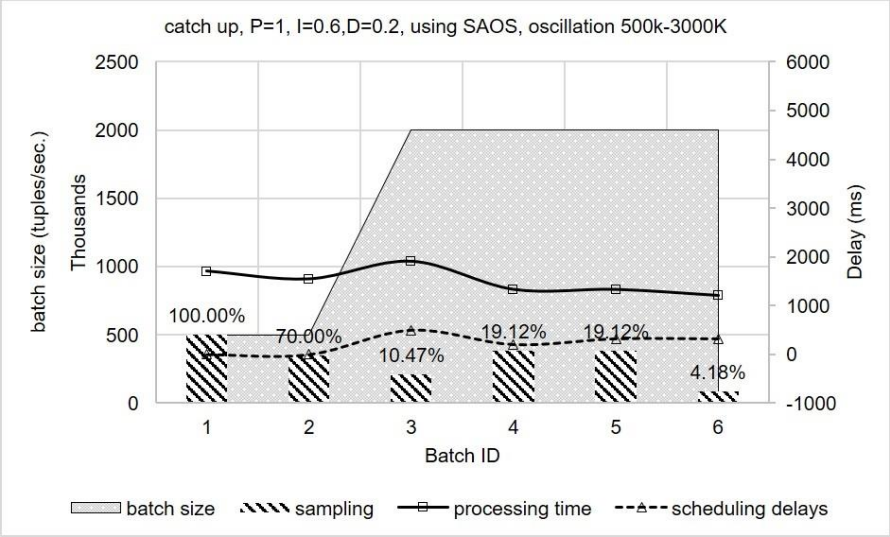

Figure 6.3. catch up at PID values 1,0.6,0.2 where SpatialSSJP is able to meet the latency target by applying the latency-aware controller. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

Figure 6.4 shows a catch-up by oscillating batch size from 500K to 3000K, hitting stronger the SPE resources. Notice that in this case, as the oscillation is higher than the previous case (being sterner, 3000K instead of 2000K) the system does not overshoot the sampling fraction, it instead keeps lowering it monotonically until the system stabilize at almost 0.02% and a plausible convergence is achieved near the latency target specified by the user. Similar trend occurs in case of PID values equal to (1,1,1), respectively as illustrated in figure 6.5.

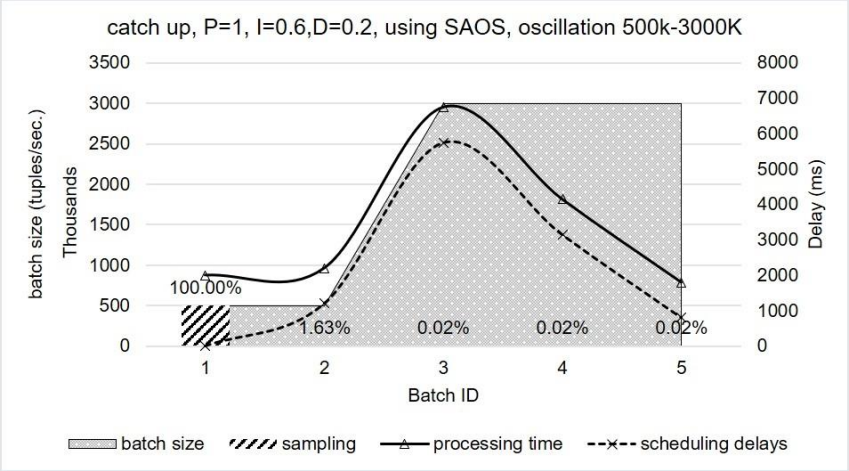

Figure 6.4. catch up at PID values 1,0.6,0.2 and oscillation 500k-3000K where SpatialSSJP is able to meet the latency target by applying the latency-aware controller. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

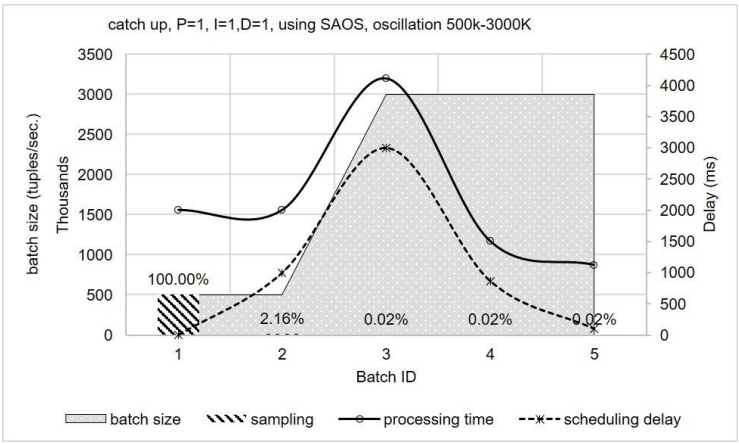

Figure 6.5. catch up at PID values 1,1,1 and oscillation 500k-3000K where SpatialSSJP is able to meet the latency target by applying the latency-aware controller. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

Being harsher, and mimicking the oscillation in wild, we mimic a sudden spike from 500K to 5000K. again, SpatialSSJP was able to catchup and stay alive for both PID settings. Figure 6.6 shows also the case where SpatialSSJP was able to survive a brutal spike in batch size, a

fluctuation happens from 500K to 2000K to 1000K. Notice how the processing/scheduling delays are smoothly and ideally following the same discernible pattern as the input rate, in a circadian rhythm, signifying the ability of the system in meeting efficiently the spikes in all ways, considering a sudden spike and a fluctuation from brutal spike to a more relaxed situation. Our method can extrapolate unseen sudden spikes in data arrival paces.

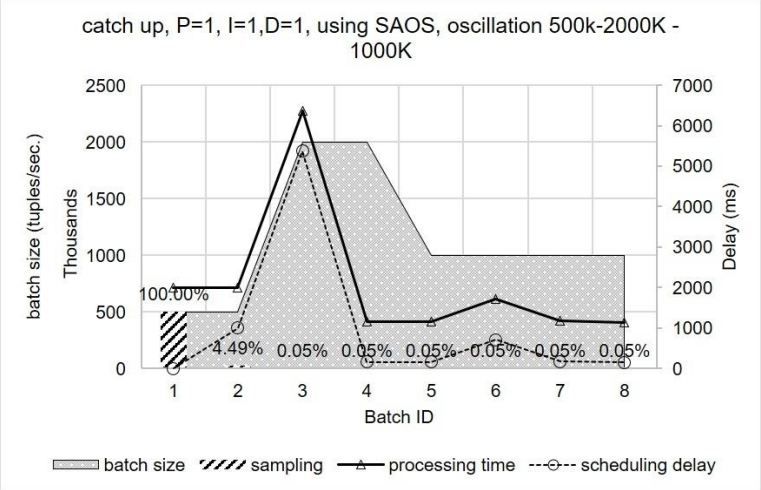

Figure 6.6. catch up at PID values 1,1,1 and oscillation 500k-2000K-1000K where SpatialSSJP is able to meet the latency target by applying the latency-aware controller, Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

We see that, at all settings, as a convergence occurs, the processing time reduces so that it falls within the batch intervals. Then, SpatialSSJP gradually starts pulling in more samples per batch. By relying on an SRS-based sampling method instead of our SAOS method, our SpatialSSJP is also able to survive spikes at all cases, however, with deteriorated accuracy bounds as it induces more Standard Errors (SE) and CV than SAOS. Figure 6.7 shows an example.

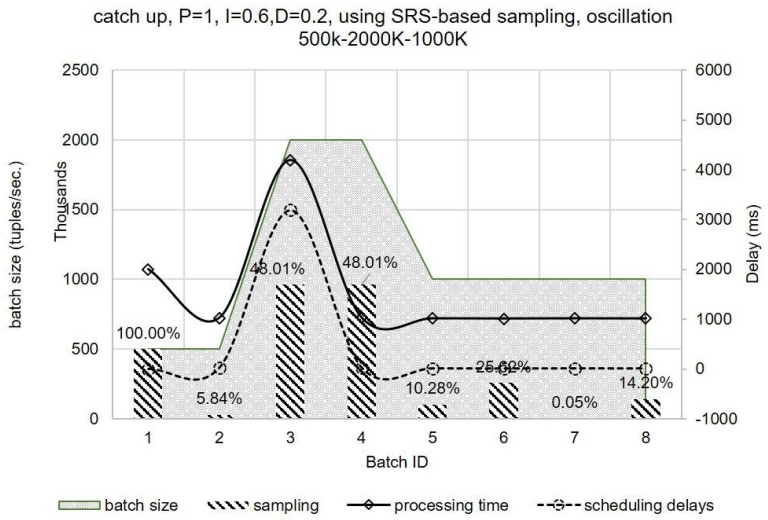

Figure 6.7. catch up (SRS) at PID values 1,0.6,0.2 and oscillation 500k-2000K- 1000K where SpatialSSJP is able to meet the latency target by applying the latency-aware controller. Secondary axis to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main axis to the left compares the batch size (data load) with sampling fraction

Despite being able to survive spikes in streaming data loads, SRS-based sampling underperforms SAOS, where the latter yields better sampling statistics in estimating target variables. This is obvious through measuring the CV as shown in figure 6.8.

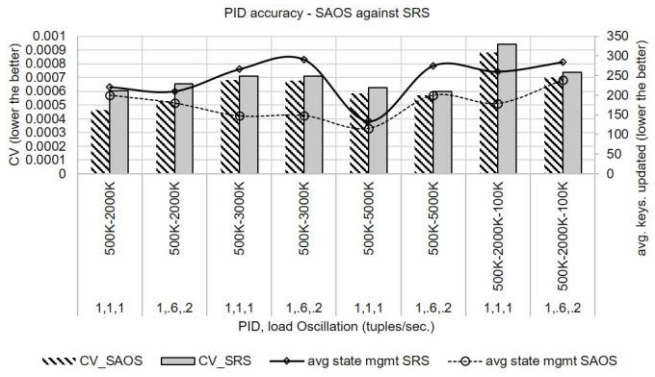

Figure 6.8 . Coefficient of Variance by applying SAOS against SRS-based, both under SpatialSSJP. 'avg state mgmt.' in the legend (corresponds to the secondary axis on the right-hand side, 'avg. keys. updated') is the average state keys managed in-between time windows. CV in the legend (corresponds to the primary axis on the left-hand side) is the Coefficient of Variance

Also, SRS-based method is susceptible to missing the confidence interval as shown in figure 6.9. The case at load oscillation (500K to 2000K), whereas, SAOS-based counterpart is perfectly residing safely in the middle, never missing the confidence interval. As shown in figure 6.10.

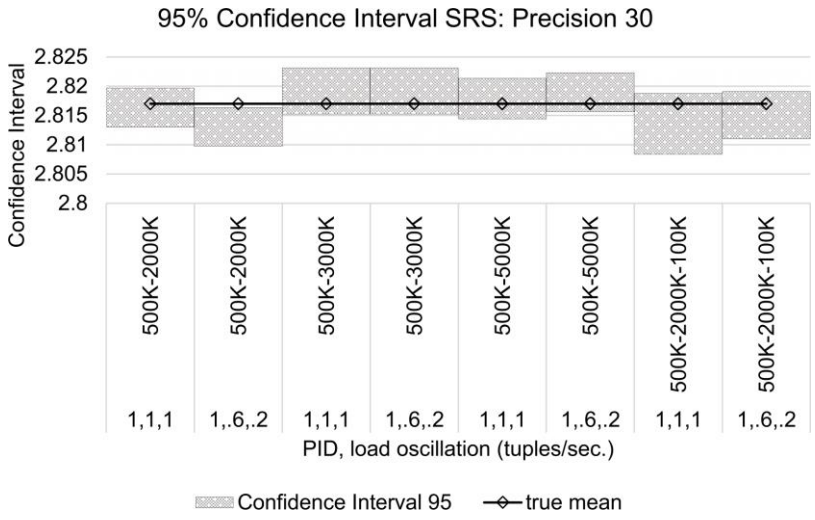

Figure 6.9. Confidence Interval true-value-miss by applying SRS with SpatialSSJP

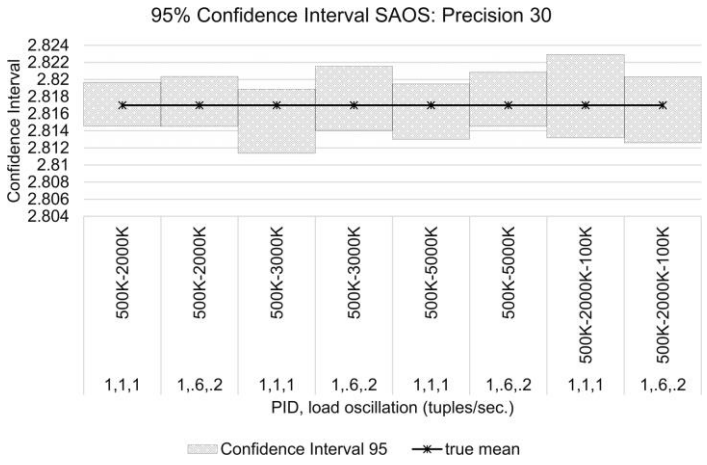

Figure 6.10. Confidence Interval true-value-always-hit by applying SAOS with SpatialSSJP

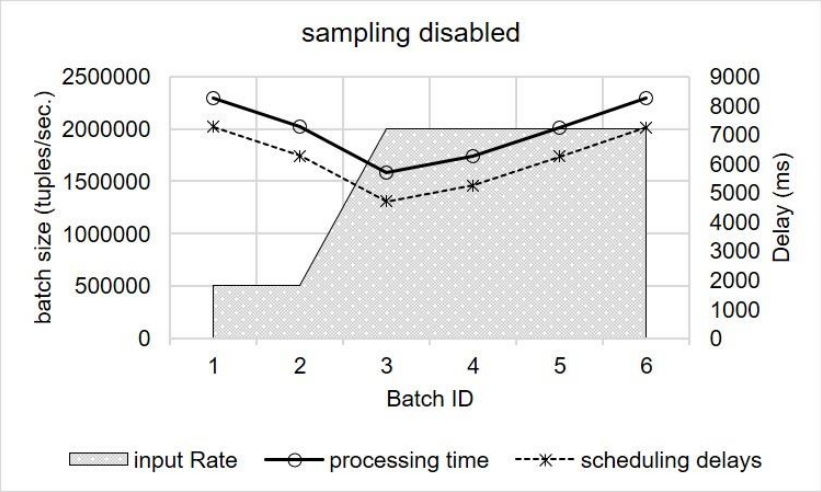

Figure 6.11. High delays imposed by disabling sampling during burst loads, Oscillation 500K – 2000K. Secondary access to the right hand-side represents 'processing time' and 'scheduling delay', whereas the main access to the left shows the batch size (input rate)

In cases where sampling is disabled, the system was not able to achieve the latency goals as shown in figure 6.11. Also, in brutal oscillation cases such as 500K to 5000K, the plain baseline system (plain spatial join operator without sampling) throws an out-of-memory (OOM) exception. Signifying the importance of applying sampling in streaming highly dynamic application scenarios.

Overall, the gain (a.k.a. precision or design effect, *deff* for short) of relying on our sampling method (SAOS from our previous work [101]) instead of an SRS-based design , and incorporating that synergistically as a front-stage quick-and-dirty sieve , is shown in figure 6.12.

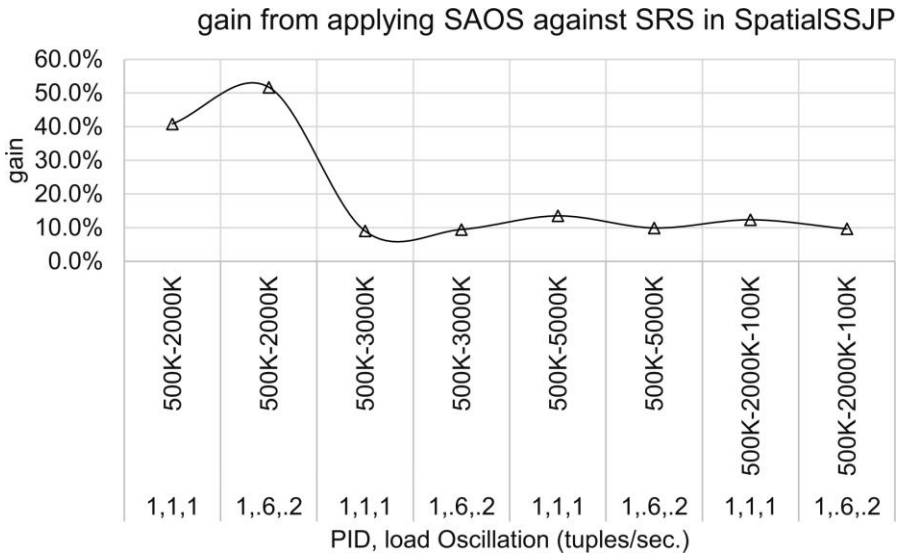

Figure 6.12. Gain by applying SAOS (with SpatialSSJP) against SRS

We can see that relying on SAOS, we achieve the best performance, significantly outperforming SRS by a large margin (on average, the relative improvement over SRS is at least 10% and reaching more than 50% at times), suggesting that relying on data-shape-aware designs (such as SAOS) is preferable over randomly selected designs.

6.5.2.2 SpatialSSJP Ability to Satisfy Accuracy (estimation quality) Target

Using our accuracy-aware rate controller, relying either on SRS-based or SAOS, we could achieve the prespecified accuracy target (i.e., estimation quality or '*margin of error'*). However, SAOS requires, on average, less sampling fractions compared to SRS-based designs in order to achieve the same error-bonded target. A plausible case, as less fractions means lower latency and higher resource utilization, thus better trading off the contradicting QoS goals. This trend is shown in figure 6.13 for two '*margin of error'* values (0.03 and a more restrictive 0.01). Notice that in more restrictive cases (i.e., when error equals to 0.01), both SAOS and SRS-based need more sampling fractions to achieve the target accuracy. However, all in all, relying on SAOS yields less sampling fractions than SRS-based, which is statistically plausible.

Because N (continuous population in each trigger) is large, n_0/N is very small, rendering $n \approx n_0$. Thus, approximately same sample is required for any large population (being 1 million or 1 billion tuples). Then it becomes readily apparent why we obtain the same almost

sampling fraction in subsequent trigger intervals (whether relying on SAOS or SRS), which corroborates the formalization herein.

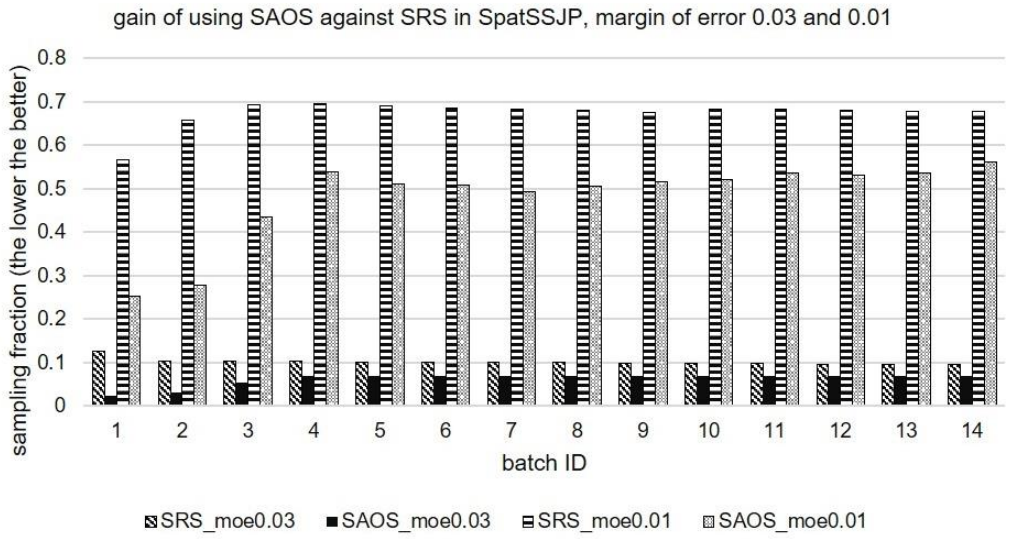

Figure 6.13. Accuracy gain by applying SAOS with SpatialSSJP against SRS. In the legend, 'moe0.03' means 'margin of error' that equals 0.03, whereas 'moe0.01' means 'margin of error' that equals 0.01

Backpressure shows similar latency improvement as our system. However, it does not reflect the latest progressions in deep insights in a timely fashion as it considers only a past time and puts a hold on the arriving data, thus negatively affecting the freshness of system output which counteracts the benefits of stream processing.

6.6 Related Work

The widespread abundance of sensor-enabled and IoT devices have catalyses the trend of data analysis to shift greatly from static views into online and real-time counterparts. As the analytics envisaged from such unbounded loads are mixed, sometimes integrating data from multiple sources, join presents itself as a main operation in any successful stream processing end-to-end pipeline.

Join is computationally expensive and can render a SPE unresponsive in burst spike workloads, where data arrival rates exceed the service rate of DAGs that encapsulate joins. To mitigate this problem, several works from the relevant literature have adopted different strategies for controlling the rate in burst streaming sources. Those systems are based on one of the approaches that we have discussed in section 6.1.3 (i.e., backpressure, elasticity and

approximate computing). Elastic approaches are classified by the method they apply in deciding when and how to adapt. Those are mainly categorized into threshold-based and model-based designs. Threshold-based elastic systems compare the size of data in a batch with a threshold, so that adaptivity (i.e., scaling resources in/out, up/down) is triggered once batch size exceeds the threshold. Just-in-time (not so early, not so late) firing of the adaptation trigger is highly desirable and can only be achieved by choosing the appropriate threshold, which is specifically challenging. Model-based approaches depend on a mathematical model to calculate when and how to adapt, however finding an expressive model that represents the system environment is challenging. Other possibilities include self-tuning learning-based models that learn statistics gradually from the data over time as time tick forward and enhance the predictivity depending, for example, on a machine learning formalization. However, those normally leave the system unstable as they involve complex machine learning models that incur additional costs which are not being amortized by the benefits they provide. Another relevant categorization of methods is being proactive or reactive. In the former, methods can predict future spikes and act as early as possible, thus avoiding any congestion, whereas in the latter methods are only reacting at the time the spike hits.

Most methods of the relevant literature focus on stream-stream joins in distributed environments. However, only little attention has been given to stream-static join processing, where data-at-flight needs to be joined with a master data-at-rest to enrich the former with appropriate descriptions [17].

Some streaming join algorithms are designed specifically to work in centralized single-device servers. They are also designed to operate in RDBMSs. For example, Wander join [112] employs graphs to model data join relationships in stream-stream settings. However, such a mechanism is not designed for distributed settings and parallelizing it is a nontrivial task.

In the same vein but applied this time to stream-static join (in what authors call semi-stream join), [118] propose a cache-based method for joining streaming data with disk-resident data under a record-at-a-time model in a centralized server-based setting. However, their adaptation mechanisms are based on complex models such as machine learning predictions

and thus add more complexities to the cost formulas, counteracting the benefits of elasticity. Again, this method is appropriate in centralized server-based settings and not readily available for distributed environments, marking its adoption non-trivial as they are unable to scale out to deal with massive data sets efficiently.

Some other algorithms, despite being designed for centralized single-node servers, focus on applying load shedding. However, most of them apply it to stream-stream joins and only few, such as [131] , apply it to stream-static join. Their approach is reactive threshold-based in the sense that they apply a simple formula for calculating the latest batch size and then if a future batch size exceeds its double, they perform load shedding to prevent accumulating tuples in the buffer. There are two distinguished problems with this approach. First, it sheds loads randomly not being attuned to the data characteristics. Also, there is an overhead incurred by continuously spilling out loads to disk and recovering them to be processed again as loads slowdown.

Aiming at parallelizing algorithms like Wander join, some other researchers tackle the distributed stream join from other angles, including the partitioning scheme. For example, [132] present a new elastic partitioning scheme for stream-stream theta-join operators, aiming at striking a balance between high throughput and high resource utilization by only acquiring resources on-demand (i.e., dynamic allocation).

So far, little attention has been given to the stream-static join processing using the micro-batch model in distributed settings. [17] propose a solution called DS-join for distributed processing of the join between streaming and stored big data under the micro-batch model of recent distributed SPEs. They focus on repartition join specifically as they target settings where the static relation does not fit in the memory of Spark worker nodes, so they aim at minimizing the shuffling. DS-join generates multiple queries that are executed in parallel using Spark Streaming.

Despite the abundance of scattered works handling joins in many directions, most of them are general-purpose and not attuned to data specific characteristics such as spatial workloads, which require specific considerations. Hence, special systems have emerged to tackle the spatial join peculiarities. From the literature, spatial-aware systems are mainly based on the batch-oriented Hadoop or the speed-oriented Spark. For example, SpatialHadoop [63] has

been built on Hadoop with appropriate indexing schemes (i.e. grid, R-tree, and R+-tree) for supporting spatial joins among other spatial queries. similarly, HadoopGIS [65] exploits Hive with a grid index for processing self-joins. So, they depend on spatial indexing to speed up pair-wise cross joins. On top of Apache Spark [1] , SpatialSpark [133] supports broadcast PIP test spatial join algorithm. From the same class, GeoSpark [11] perform spatial joins by indexing on quadtree and R-tree for local indexing, whereas employing regular grid for global indexing, hence resembling two-level indexing. However, there are few apparent limitations with those systems. First, they are basically designed to support static-static spatial joins. Second, they do not natively support SQL-like queries. Moreover, they do not incorporate approximate methods for handling burst workloads in case they are retrofitted to work with streams.

We are not aware of any works in the relevant literature that exploit approximate processing (using sampling basically) to support stream-static joins specifically for spatial workloads. However, some works apply sampling for general streaming workloads in burst environments. For example, [134] propose an adaptive overload management system *AccStream* (on top of Spark Streaming [22]) which selectively samples/drops and processes data tuples (and sometimes blocks, building blocks in Spark Streaming terms) on a de facto mini-batch streaming SPE. *AccStream* consists of three elements; a *controller*, *collector* and a *retrofitted receiver*. The *collector* sends statistical information (i.e., latency and accuracy, accuracy depends on sampling theory) to the *controller*, that, in turns, computes an appropriate sampling fraction. The *receiver* is a retrofitted version of Spark Streaming's receiver so that it incorporates a sampling module that samples at the granularity of tuples and blocks. For achieving the latency targets, they employ dynamically a self-tuning learning-based model (i.e., latency model). The downside however is that *AccStream* is general purpose and not readily prepared for spatial loads. Also, it only supports two analytics, single aggregations (such as 'counts') and top-k. This also implies other statistical estimates that are composable of those, such as 'averages' (i.e., 'means') that are composable of two divided sums. Also, the complex method they are using for system prediction endangers the system stability. This is because the method that they have described is computationally expensive and requires continuously calculating many statistics that are not

readily available by the underlying system codebase (i.e., Spark in this case), rendering the model as a bottleneck that can carry more latency in subsequent time windows.

We are not aware of any system from the relevant literature that achieves the goals we aim at achieving by designing SpatialSSJP.

To summarize the relevant art, most works presume a (nearly) perfect envisaged knowledge of the future. However, an online algorithm that simply self-tunes and does not have complete knowledge of the future is more desired [115]. Also, most approaches, despite being able to maximally utilize resources at times, need to be manually tuned with specific workloads at most other times. They also have limitations in handling live data streams and poorly model QoS requirements [127]. Moreover, they are not intrinsically designed to handle geospatial workloads that normally show temporal skewness in intensities. Those are some reasons that have encouraged us to design SpatialSSJP, most importantly, considering a controller such as PID that does not expect too much knowledge of the future and being able to keep the system stable. Also, successfully applying methods from the sampling theory in modelling an accuracy aware controller, thus efficiently modelling QoS latency and accuracy requirements that are prespecified by an expert user.

6.7 Chapter Conclusion

Elastic scaling of resources has been thus far the predominant solution for surviving in transient burst spikes of streaming data loads. Aiming basically at maximizing the resource utilization by (semi-)automatically provisioning and deprovisioning resources. Also, threshold-based and other models require a manual tuning of the configurations which may need specific domain or workload knowledge and prediction. Only little knowledge, or no knowledge at all should be envisaged in those settings.

The highly skewed nature of spatial loads requires careful attention. It also requires being attuned to those characteristics in order to be able to handle spikes and oscillations in spatial data arrival rates in streaming deployments.

To close those gaps in the literature, we have designed an adaptive spatial aware processing engine, that most importantly focus on the stream-static joins (better known as geofencing or PIP for spatial loads). Our system proves efficient as it can strike a plausible balance

between contradicting QoS harsh constraints (such as latency and accuracy or estimation quality). It does so by employing two entwined controllers, an accuracy-aware controller that carefully obeys the sampling theory and a stable loop-feedback mechanism (PID) that keeps the system stable by not undershooting or overshooting the sampling fractions, with a minimal calculation effort that only adds negligible costs that are mitigated by the benefits we reap from the adaptation. Our system, SpatialSSJP is able to rejuvenate the operation of the join operator even after an overwhelmingly striking blazing-fast spikes in data arrival rates. We do so by only tuning one parameter, the sampling fraction. In addition, our system is the first in its class that adopt an SQL-like declarative API for SAQP (specifically for stream-static join processing) by being built on top of SpSS, thus exploiting all the query optimizations that are provided already by the codebase. In addition, although designed to work with micro-batching systems, it can be easily extended to other SPEs that support other window semantics. Results on large-scale datasets show that SpatialSSJP cultivates a significant improvement over baselines.

As a future research perspective, we have only considered the cadence of data arrival rates as the major cause of latency in current SPEs. However, there are many other causes that may be detrimental to the overall health of the SPE. Most significantly, perhaps is the cross-network shuffling being a potential confounder, which is basically caused by employing naïve partitioning schemes that are unaware of the spatial characteristics.

Chapter 7

Conclusion and Future Works

In this chapter, we first summarize the major contributions that we have made in this thesis in § 7.1. In what follows, we explain in § 7.2 the wide range of applicability of the contributions of our systems and algorithms in diverse domains, specifically for highly dynamic and scalable application scenarios. To conclude the chapter in § 7.3, we recommend interesting future research frontiers that can be based on the primitives and baselines we have presented in this thesis.

7.1 Summary of Contributions

Avalanches of geospatial data that are streaming from various, often, heterogeneous channels are looming threats on businesses and presenting them with formidable challenges and hazards. In addition to the significant patterns that are hiding deeply inside stockpiles of geo-referenced data. Neither streaming data nor batch snapshots can exist in void, they are complementing each other and analyzing each of them alone is not revealing the whole picture that can assist better decision making. We posit that “one-size-fits-all” does not hold true in distributed spatial stream processing and management environments. Often, historical deep insights need to be combined with data-in-motion so as to improve the analytics quality.

Current systems do not natively offer QoS awareness as a transparent underlying layer for processing streams of geo-referenced data. More than often, users need a technical knowledge to tune at the QoS level. A QoS aware system for processing fast arriving spatial data streams is then needed, which transparently incorporates QoS awareness within its layers so that it constituent parts operate synergistically in an aim at achieving a prespecified set of QoS goals. This consequently means that the users at the presentation layer do not need to reason about the underlying QoS logistics, but otherwise use them in their applications seamlessly.

To achieve those goals and close the gaps in the literature, in this thesis, we have designed a QoS Aware DSMS for geo-referenced huge amounts of streaming data loads (we term our

system as SpatialDSMS. The system is built with a modular architecture that streamlines the orchestration between the constituent sub-systems such that the development and deployment efforts are not repeated for every workload alone. Instead, the systems we have designed and incorporated in SpatialDSMS work collaboratively and synergistically in achieving the modularity. Colloquially, traditional independent big geospatial management systems are operating in an uncharted territory, and SpatialDSMS is the compass. It has been designed to provide an unrivalled capacity to achieve desired QoS goals intrinsically. We have specifically designed, implemented and incorporated in SpatialDSMS the following sub-systems:

7.1.1 SpatialBPE

SpatialBPE is the part that is responsible for batch processing of the arriving workloads in batch mode. This means that snapshots of streaming data are first spilled to disk. Thereafter, on need, SpatialBPE could be asked to analyze portions from this data-at-rest to get some historical insights that assist in decision making. The QoS of this component depends on its ability in serving results faster at times (i.e., low-latency QoS goal). Also, it is desirable to localize the geographically-nearby spatial objects so that to minimize network shuffling and thus allowing for a QoS aware sharing of network resources, thus achieving the high resource-utilization QoS goal. Those QoS aware services are transparently injected on top of the codebases of best-in-class representatives (i.e., Spark in this case). Having done that, SpatialBPE assists in complementing the modular architectural design goal that has been envisaged by designing SpatialDSMS.

7.1.2 SpatailNoSQL

SpatialNoSQL constitutes a scalable backend QoS aware storage framework for geo-referenced streaming data snapshots. It is consolidating heterogeneous resources in a unified compatible format. Snapshots coming from streams are transformed into that format and sharded appropriately (i.e., depending on QoS aware rules) to multiple shards in such a way that assists in achieving QoS goals prespecified by the user. SpatialNoSQL constitutes a custom sharding scheme (i.e., GSS) that is attuned to the data shape (i.e., being spatial). It then helps in striking a plausible balance between two sharding goals (i.e., SDL preservation and load balancing). It also hosts two spatial query optimizers that exploits our custom

sharding scheme in achieving the QoS goals. Being designed to complement the other components of SpatialDSMS, it has a modular architecture that enables it synergistically to co-work with the other components to solve mixed workloads problems. For example, for a fast approximate stream-static join , since the static table is stored in SpatialNoSQL with polygons represented as covering geohashes, then it would be easy to combine with a geohashed streaming data load as we simply need to overlay the streaming points map (from a micro-batch) on the covering polygons map and the join is solved in a simpler way known as MBR-join , acting as a quick sieve with statistically rigorous error bounds. The fact that both geospatial objects (i.e., the stream and the static master table) have the same representation (i.e., geohash) has enabled this kind of mix workloads with QoS guarantees. This also has encouraged us to design SpatialSPE, which is discussed in the next subsection.

7.1.3 SpatialSPE

For huge spikes that need to be processed fast, where we can sacrifice tiny accuracy for huge performance gains (i.e., low-latency, high-throughput and high-resource-utilization), we have designed SpatialSPE as the first in its class that is able to perform incremental spatial analytics based on a declarative SQL-like API, thus relieving the shoulders of geo-statisticians from having to reason about the intricacies and complexities of the underlying systems and focusing instead on the statistical analytics part. SpatialSPE is based on robust statistical modelling and is implemented with an emerging micro-batch streaming SPE (i.e., Spark Structured Streaming). SpatialSPE hosts a spatial-aware sampling method SAOS, which is attuned to the data characteristics. Thus, we reap many benefits that efficiently impact the QoS goals. SpatialSPE is able to achieve statistically plausible results and by orders-of-magnitude outperforms its counterparts. SpatialSPE complements the modularity architectural design goal of SpatialDSMS in the sense that it incorporates seamlessly with other components so that they all synergistically and collaboratively achieve an envisaged set of QoS goals.

7.1.4 SpatialSSJP

Most interesting insightful analysis happen during the spike in streaming data arrival rates, which, at times, necessitates mixing the fast loads with disk-resident descriptions, in a costly operation that is mostly known as stream-static join. We have designed SpatialSSJP so that

it complements the other components of SpatialDSMS in modular way. SpatialSSJP incorporates QoS aware services transparently within the layers of codebases of best-in-breed SPE (i.e., SpSS) so as to relieve the overburdened shoulders of the users at the presentation layer from having to reason about the underlying complex logistics. Services include an adaptive controller that constitutes two sub-controllers, one that is latency-aware based on the PID from the control theory and the other one is a model-based accuracy aware controller that is based on geo-statistical modelling. SpatialSSJP is modular by design and conveniently complements the other components of SpatialDSMS. Most importantly, it reuses our SAOS sampling method from the SpatialSPE framework.

7.1.5 Putting it All Together: SpatialDSMS

Dynamic applications in smart cities and Industry 4.0 require mixing several workloads so as to get deeper insights. The constituent parts of SpatialDSMS provide tools for

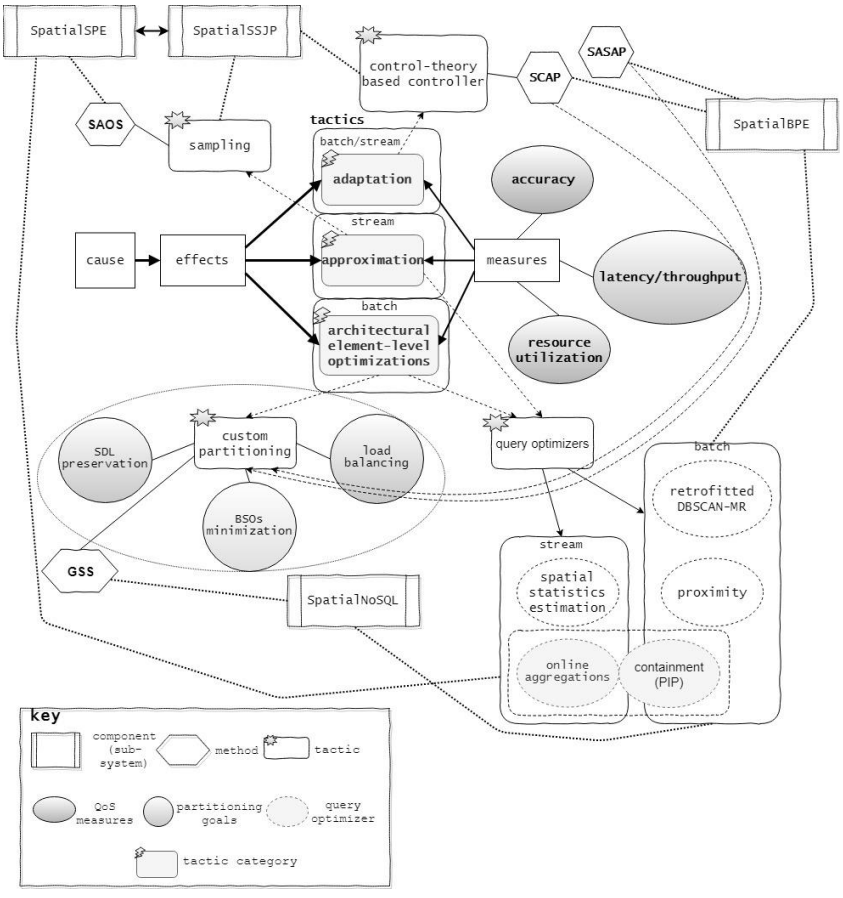

Figure 7.1. SpatialDSMS contributions map

collaboratively and synergistically achieving QoS goals imposed by those workloads. QoS awareness is transparently incorporated within various layers of SpatialDSMS, thus relieving the shoulders of the users from having to reason about the underlying logistics for handling such awareness.

The map of figure 7.1 delineates in a coherent way the contributions we have made in this thesis and all the tactics and methods we have designed for achieving a list of envisaged QoS goals. This map complies with the methodology we have designed as described in section 3.2.1.

7.2 Applicability of SpatialDSMS in Diverse Domains

QoS -aware optimizations we have provided in this thesis are in no way exhaustive, instead they constitute precursors for other domain-specific optimizations. One of the design goals that we have envisaged by designing SpatialDSMS is the *composability* (refer to section 3.4.1 for further details). It is loosely defined as the ability to use the primitive QoS aware services that we have provided in SpatialDSMS in order to serve other potential workloads that are common in highly dynamic and scalable applications. In this section, we recapitulate some mixed workloads that are easily composable by mixing some of the services we provide through SpatialDSMS. The following is a non-exhaustive list of emerging trendy applications for which we provide efficient and sufficient QoS-aware baseline primitives (in addition to other easily composable primitives) that allow constructing novel highly-performing algorithms.

I) **Real-time traffic control**. We are not offering an engine that over-the-counter supports such scenarios as they require specific technical implementations. For example, calculating *traffic flow correlations*, *indicators* [135] , *flow rates*, *occupancy* and *density* and others that can be consulted in [136]. We offer baselines (i.e., spatial statistics) that can be adequately exploited by most emerging smart traffic control systems (e.g., [26, 135]) to build a fully-functional (near) real-time traffic control system. Despite our system does not calculate those measures directly, it offers appropriate and adequate baselines and primitives that can be used seamlessly to calculate those measures, with a distinction from

counterparts, in a way that better achieves a prespecified set of QoS goals. Another distinction is that we support incrementalization for a primitive set of spatial statistics that can be exploited efficiently in achieving those operations. Those services are offered through SpatialSPE and SpatialSSJP specifically.

II) **Spatial online stream clustering**. An interesting mixed workload could ask to interactively "disseminate targeted warnings to people in real-time in cases of sudden natural hazards, such as hurricanes". Hence, the dynamic identification of homogeneous clusters in (near) real-time is essential. For example, referring to our scenario (section 1.1) we can dynamically partition an embedding space (i.e., region) into smaller regions (such as boroughs or districts in administrative management terms) around a hazardous situation by exploiting real-time streaming data clustering to assist in emergency response management. Personalized notifications can then be forwarded to each cluster independently. Similar application scenario is ascribed to [137].

Although, we currently do not support online clustering over-the-shelf, it is easily *composable* from two clustering modes that we provide natively. Most online clustering algorithms work by combining two phases (e.g., CluStream [138] and DenStream [76]), online and offline clustering phases. The former applies a single pass scan over a fast arriving data stream to *incrementally* cluster data points based on proximity, thus forming *micro-clusters* that basically store streaming online aggregates (i.e., statistics such as the number of points in each cluster in addition to other summary statistics, such as 'sums' and 'counts'). This is accomplished incrementally by either assigning each newly arriving point to its appropriate *micro-cluster* (i.e., based on a spatial proximity test) or creating a new micro-cluster. The size of the cluster (thereby the number of micro-clusters) is a tunable threshold [137] [139]. The offline *macro-clustering* phase is a costly operation that forms the actual clusters and is normally performed in batch mode (i.e., offline), after it receives the list

of micro-clusters, it uses them in conjunction with other parameters to construct the actual clusters by using an advanced clustering algorithm such as DBSCAN (or its MapReduce-based variants in parallel settings, such as DBSCAN-MR).

The *composability* of a novel spatial online stream clustering algorithm with quality guarantees is easy and straightforward by using the baseline primitives we support through SpatialDSMS. Specifically, our optimized version of DBSCAN-MR (discussed in section 4.7.5) for the offline macro-clustering phase. Also, our approximate spatial query processing , that streamlines the process of collecting in piecemeal fashion the online summary statistics by applying a lower dimensional index structure (i.e., based on z-curves) , thus dynamically on-the-fly clustering and forming micro-clusters (as of yet not the final clusters), as discussed in chapter 5. As the micro-clustering phase is the online statistical data collection portion of the algorithm, we have shown how our SpatialSPE (explained in chapter 5) is adept in such a process, serving statistically plausible incremental results with rigorous error bounds.

7.3 Future Works

In this thesis, we have presented SpatialDSMS as a comprehensive QoS-aware architecture for optimized analytics of spatial data loads in highly dynamic application scenarios that, among other functional QoS goals, require scalability. We have addressed many challenges in en-route to striking a plausible balance between a list of contradicting QoS goals. The way we incorporate transparently our QoS-aware services into the layers of SpatialDSMS is unique. To the best of our knowledge, we are not aware of any similar system that achieves goals similar to those that we have accomplished in SpatialDSMS.

However, SpatialDSMS is not a panacea, and there are innumerable ways in which our modular architecture can be complemented by stacking up new modules that achieve QoS functional and non-functional goals. We here list some possible future research frontiers:

1) **Offloading** sequential jobs to Fog nodes. The communication overhead posed by sending endlessly huge amounts of geo-referenced loads to the cloud which could be

detrimental in low-latency applications. Especially knowing that some parts of the work are sequential and do not need parallelization (or cannot run in parallel). For example, partitioning data. Those sequential portions of the work can be offloaded to Fog nodes in an efficient way that considers the resource-constrained nature of Fog nodes. Also, samplers (such as our sampler SAOS from the SpatialSPE) can be pushed upstream near the Edge, which potentially helps in achieving better latency QoS goals.

2) **Designing** online spatial-aware data partitioning schemes. We did not consider spatial-aware data partitioning schemes. As a future frontier, on-the-fly schemes and indexing are needed to strike a balance between SDL preservation, BSOs minimization and load balancing for the data in-motion. taking those goals online enforces few challenges that do not affect batch partitioning schemes such as those that we have addressed for spatial batch processing systems.
3) **Designing** distributed sampling methods. Our sampling is currently centralized, performed by a single node as a front-stage. One way for optimizing that is to design a distributed sampling approach that parallelizes the sampling portion of the equation, thus enabling more performance optimization in compliance with the Amdahl's Law. This is to avoid the cases where the sequential centralized solution can become itself a bottleneck. We envision a multi-stages scheme, say macro- and micro-batching stages. In the macro-batching stage, a practitioner (i.e., could be hosted in a master node) forms macro-batches and emits each *macro-batch* to a worker node, which in turns, divides the macro-batch into *micro-batches* and distribute them efficiently.

List of Figures

List of Tables

List of Algorithms

List of Listings

Appendices

Appendix A

GeoSpark Architecture

GeoSpark [11] consists of three layers stacked up in a tiered architecture; i) Apache Spark codebase, ii) spatial RDD and iii) spatial query optimizers, arranged in a bottom-up layered pattern, respectively. GeoSpark provides four new spatial data structures based on RDDs, PointRDD, RectangleRDD, PolygonRDD and CircleRDD. It supports geometric operations on each of them and also provide spatial indexing structures such as quadtree [51] and R-Tree [140]. Top layer is responsible for executing spatial queries over large scale geo-referenced datasets. After creating a spatial RDD, it is imposed to the spatial query predicates and optimizers are responsible for computing answers and serve them to presentation layers thereafter.

Appendix B

DBSCAN-MR Workflow

In short, DBSCAN-MR proceeds as follows. Local clusters are formed by applying the plain DBSCAN to each partition independently. Most operations involved are '*map'* transformations. Once the algorithm have done examining all points in all nodes, the output of the '*mapping'* returns a new RDD, this time with the key ID of the point (specifying to which partition it belongs) and the point object (a module that we have defined to reformat points). Afterwards, local clusters (we refer to them as micro-clusters) from independent partitions are emitted to a '*reduce'* phase in the DAG network. The '*reduce'* function then groups together all elements that share the same ID (which were replicated on multiple partitions), which determines the union of temporary clusters located in different partitions that will be merged in a later stage. Results from the '*reduce'* phase are merged to find out the cluster's global structure. The algorithm concludes by applying a relabeling phase, where each core local point that belongs to a global cluster (but residing in independent partitions) is relabeled to identify the resulting cluster.

Appendix C

Calculating Throughput in SpatialSPE

Throughput. (refer to section 3.2 for a wider generic definition). For SpSS, we simply calculate the throughput by counting the number of tuples that our system could process in every time-based window interval (a tumbling window in our settings). To achieve that, we employ the 'StreamingQueryListener' (a module readily available from SpSS) to capture '*start'* and '*end'* timestamps and '*number of processed tuples'*. Thereafter, we apply a simple formula that divides the '*number of processed tuples*' by the total time elapsed during a continuous query window (a tumbling window in our settings).

Appendix D

Spatial Sampling Distributions: Data Skewness

Despite that the NYC taxicab dataset is highly skewed. The average (mean estimator) has an approximately normal distribution (informally, bell-shaped curve) in sampling distribution. In accordance with the Central Limit Theorem (CLT) [90] , principles from traditional statistical sampling applies, specifically those that are coming from classical stratified and probability sampling theories. Figure D.1 shows that despite population data is highly skewed, a sample of 1000 '*means'* of few values is normally distributed, also by increasing the sample size the distribution becomes more normalized. Notice that in compliance with the normal distribution theory [90] , from the cohort data that we have chosen, for the '*means'* calculated for 10 and 40 values repeated 1000 times, 68%, 95% and 99.7% of data falls within at most one, two and three '*standard deviations'* farther from the '*mean'* value, respectively, which further supports the applicability of default general sampling theories [90].

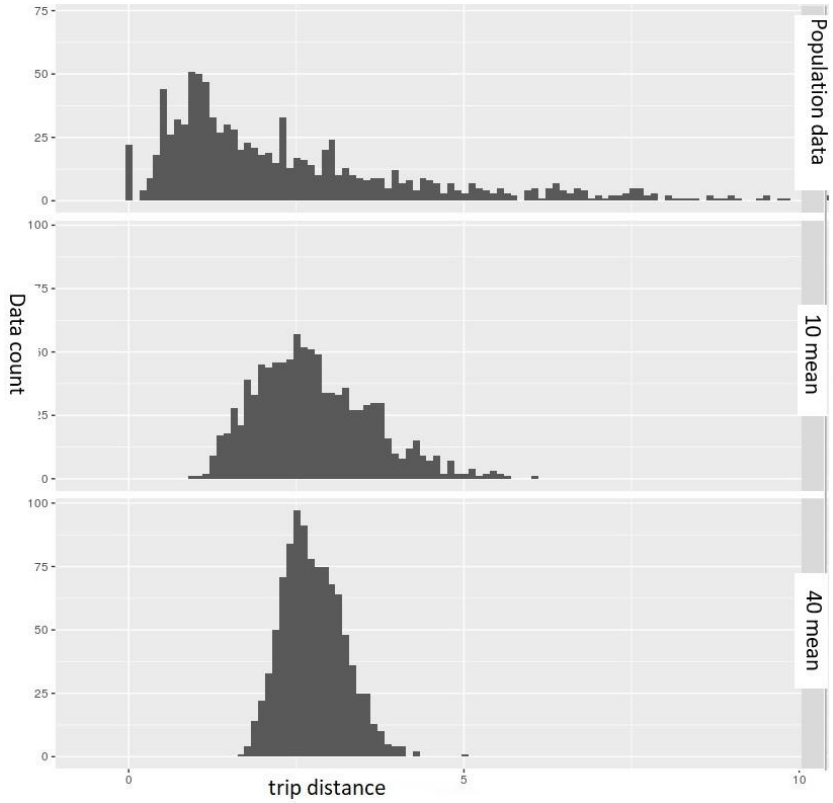

Figure 0.1. popultion data distribution, and sampling distribution for the means of 10 and 40 values, respectively, repeated 1000 times.

Appendix E

Further Words on SAOS Efficiency: Theoretical Perspectives

Reiterating our canonical scenario with NYC taxicab (from section 5.3). Imagine the desire to answer the following question, "where do people tend to order taxi pickups in NYC".

In SpSS, using a fraction of the data stream, this can be expressed using the fluent API with the CQ shown in listing E.1.

```
sample= RawStream.SAOS("geocode")

sampleTransformed =
sample.groupBy($"neighborhood").count().orderBy($"count").desc
with error-bound

continuosQuery=
sampleTransformed.writeStream.trigger(ProcessingTime).start()
```

listing E.1. An example continuous query in Scala-like format by using SAOS method

Where SAOS is our spatial-aware online sampling method described in section 5.3.4. Since the default mode of operation in SpSS is micro-batching, this compiles down to a traditional Spark job that is composed of a DAG of independent tasks [6, 16]. The math behind the transformation of the foregoing query is then flattened into a '*selection'* (as our SAOS method depends on a '*filter*' transformation in addition to other '*Map'* tasks). This is followed by stateful aggregations (groupBy, *orderBy*) which execute as '*reduce*' tasks, where the '*reduce'* tasks is self-informed about the in-memory state on workers and checkpoint that to a '*persistent state store*' every trigger. This can be schematically illustrated in block diagram of figure 0.2.

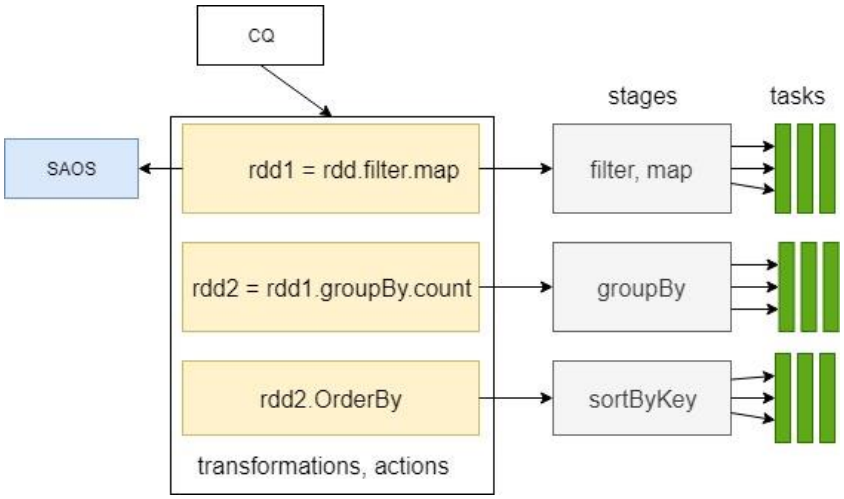

Figure 0.2. Internals of a CQ (listing E.1) incorporating SAOS

This design reveals the fact that extra a-priori overhead carried by our patches is minimal, as it mainly depends on relatively cheap '*map'* and '*filter'* transformations as a long-lived front-stage running lazily over all micro-batches for each trigger. Our method acts as a quick-and-clean sieve that ensures that we do not overlook specific study regions. Those patches do not materialize data. Instead, they engage as a low-cost stage preceding any '*reduce*' tasks. This is also possible because we treat each stratum independently, where we apply a random sampling for each stratum. Even dispatching a Spark's streaming job to worker nodes at each batch interval does not affect the '*embarrassingly parallelism*' of our design, which is naturally massively parallelizable. Our method, when translates down to a query plan (extending those offered by Spark's SQL optimizers) divides each job into tasks and disseminates them to partitions (a task for each), where each task acts on a single partition

hosted by a worker node independently for the sampling stage. The logic behind this is self-explanatory, where internally, our method acts on a for-each-partition basis, where we select a known fraction from each stratum in each partition according to a pre-defined sampling fraction map. Those independent tasks do not need to interplay, and hence no costly shuffling is introduced at this stage.

Appendix F

PID controller calculations similar to the way it has been used for backpressure in Spark Streaming [22, 126] .

After each trigger, the new rate is calculated with (F.1), adapted from the Spark Streaming [22] PID rate estimator. This PID controller has been retrofitted and transparently incorporated with SpSS layers so that it serves back new sampling rates to samplers (in this case, our sampler SAOS or an SRS baseline) in the frontstage.

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

Where $rate_{new}$ is the new rate calculated after each trigger (i.e., batch interval in Spark Streaming version), err is the difference between the desired rate (i.e., desired setpoint (SP) in PID original jargon) and the measured rate (PV in PID original terms) based on information collected from the most recent trigger (i.e., batch in Spark Streaming terms). $rate_{latest}$ constitutes the desired rate, readily available from previous trigger running information (free of charge as SpSS provides this information by default). $rate_{process}$ is then the measured process variable (i.e., PV), err is then given by (F.2).

$$err = rate_{latest} - rate_{process} \quad \text{(F.2)}$$

The integral term is used as an indicator to the historical error, specifying the amount of load that could not be processed in all the previous triggers (i.e., batches), leading to delay. We depend on (F.3) in calculating err_{hist}.

$$err_{hist} = delay_{sched} . rate_{process} / batchInterval \quad \text{(F.3)}$$

The derivative term predicts the future as the error change between two triggers (i.e., the trend), we depend on (F.4) for calculating err_d.

$$err_d = (err - err_{latest}) / (time_{current} - time_{latest}) \quad \text{(F.4)}$$

Bibliography

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, *(10-10)*, pp. 95, 2010.

[2] K. Banker, *MongoDB in Action.* Manning Publications Co., 2011.

[3] I. M. Aljawarneh, P. Bellavista, C. R. De Rolt and L. Foschini, "Dynamic identification of participatory mobile health communities," in *Cloud Infrastructures, Services, and IoT Systems for Smart Cities*Anonymous Springer, 2017, pp. 208-217.

[4] I. M. Al Jawarneh, P. Bellavista, F. Casimiro, A. Corradi and L. Foschini, "Cost-effective strategies for provisioning NoSQL storage services in support for industry 4.0," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 1227.

[5] Y. Zheng, "Urban computing: Tackling urban challenges using big data," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, 2016, pp. 3. DOI: 10.1109/RE.2016.14.

[6] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 601-613.

[7] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems.* New York; Manning Publications Co., 2015.

[8] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* " O'Reilly Media, Inc.", 2013.

[9] S. Bradshaw and K. Chodorow, *Mongodb: The Definitive Guide: Powerful and Scalable Data Storage, 3rd Edn.* O'Reilly Media Inc, USA, 2018.

[10] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The hadoop distributed file system." in *Msst*, 2010, pp. 1-10.

[11] J. Yu, J. Wu and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2015, pp. 70.

[12] R. Sriharsha. "*Magellan: Geospatial Analytics Using Spark"*. 2015. Accessed on: August 5, 2019. [Online]. Available: https://github.com/harsha2010/magellan.

[13] R. Sriharsha, *"Magellan: geospatial analytics on spark,"*. October 2015. Accessed on: August 5, 2019. [Online]. Available: https://blog.cloudera.com/magellan-geospatial-analytics-in-spark/

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 2.

[15] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark.* " O'Reilly Media, Inc.", 2017.

[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin and A. Ghodsi, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1383-1394.

[17] Y. Jeon, K. Lee and H. Kim, "Distributed Join Processing Between Streaming and Stored Big Data Under the Micro-Batch Model," *IEEE Access*, vol. 7, pp. 34583-34598, 2019.

[18] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann and A. Kemper, "Adaptive geospatial joins for modern hardware," *arXiv Preprint arXiv:1802.09488*, 2018.

[19] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann and A. Kemper, "Approximate geospatial joins with precision guarantees," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1360-1363.

[20] T. Brinkhoff, H. Kriegel, R. Schneider and B. Seeger, *Multi-Step Processing of Spatial Joins.* ACM, 1994*23(2).*

[21] A. Arasu, S. Babu and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, *(2)*, pp. 121-142, 2006.

[22] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 423-438.

[23] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen and V. Markl, "Benchmarking distributed stream processing engines," *arXiv Preprint arXiv:1802.08496*, 2018.

[24] M. A. Lopez, A. G. P. Lobato and O. C. M. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016, pp. 1-6.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 423-438.

[26] S. Amini, I. Gerostathopoulos and C. Prehofer, "Big data analytics architecture for real-time traffic control," in *2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, 2017, pp. 710-715.

[27] C. Junghans and M. Gertz, "Modeling and prediction of moving region trajectories," in *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*, 2010, pp. 23-30.

[28] J. C. Whittier, Q. Liang and S. Nittel, "Evaluating stream predicates over dynamic fields," in *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming*, 2014, pp. 2-11.

[29] J. Bao, Y. Zheng, D. Wilkie and M. Mokbel, "Recommendations in location-based social networks: a survey," *GeoInformatica*, vol. 19, *(3)*, pp. 525-565, 2015.

[30] H. Abdelhaq and M. Gertz, "On the locality of keywords in twitter streams," in *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming*, 2014, pp. 12-20.

[31] A. Pozdnoukhov and F. Walsh, "Exploratory novelty identification in human activity data streams," in *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*, 2010, pp. 59-62.

[32] H. Wei, J. Sankaranarayanan and H. Samet, "Measuring spatial influence of twitter users by interactions," in *Proceedings of the 1st ACM SIGSPATIAL Workshop on Analytics for Local Events and News*, 2017, pp. 2.

[33] P. Wang, X. Li, Y. Zheng, C. Aggarwal and Y. Fu, "Spatiotemporal Representation Learning for Driving Behavior Analysis: A Joint Perspective of Peer and Temporal Dependencies," *IEEE Trans. Knowled. Data Eng.*, 2019.

[34] M. Jensen, J. Gutierrez and J. Pedersen, "Location intelligence application in digital data activity dimensioning in smart cities," *Procedia Computer Science*, vol. 36, pp. 418-424, 2014.

[35] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Transactions on Database Systems (TODS)*, vol. 32, *(1)*, pp. 7, 2007.

[36] H. Kriegel, P. Kröger, J. Sander and A. Zimek, "Density-based clustering," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, *(3)*, pp. 231-240, 2011.

[37] M. Ester, H. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd,* 1996, pp. 226-231.

[38] B. Dai and I. Lin, "Efficient map/reduce-based dbscan algorithm with optimized data partition," in *2012 IEEE Fifth International Conference on Cloud Computing,* 2012, pp. 59-66.

[39] Y. He, H. Tan, W. Luo, S. Feng and J. Fan, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *Frontiers of Computer Science,* vol. 8, *(1),* pp. 83-99, 2014.

[40] R. Xu and D. Wunsch, *Clustering.* John Wiley & Sons, 2008.

[41] W. Kim, Y. Kim and K. Shim, "Parallel computation of k-nearest neighbor joins using MapReduce," in *2016 IEEE International Conference on Big Data (Big Data),* 2016, pp. 696-705.

[42] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref and A. K. Elmagarmid, "Incremental evaluation of sliding-window queries over data streams," *IEEE Trans. Knowled. Data Eng.,* vol. 19, *(1),* pp. 57-72, 2006.

[43] J. Kreps, N. Narkhede and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB,* 2011, pp. 1-7.

[44] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, 2011.

[45] D. Taniar, C. H. Leung, W. Rahayu and S. Goel, *High-Performance Parallel Database Processing and Grid Databases.* John Wiley & Sons, 2008.

[46] P. Furtado, "A survey of parallel and distributed data warehouses," *International Journal of Data Warehousing and Mining (IJDWM),* vol. 5, *(2),* pp. 57-77, 2009.

[47] H. Karau, A. Konwinski, P. Wendell and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis.* " O'Reilly Media, Inc.", 2015.

[48] H. Samet, "Multidimensional spatial data structures," in *Handbook of Data Structures and Applications*Anonymous Chapman and Hall/CRC, 2018, pp. 251-275.

[49] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Computing Surveys (CSUR)*, vol. 11, *(4)*, pp. 397-409, 1979.

[50] D. E. Knuth, "The art of computer programming: Sorting and Searching, 2nd edn., vol. 3," 1998.

[51] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, *(1)*, pp. 1-9, 1974.

[52] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun ACM*, vol. 18, *(9)*, pp. 509-517, 1975.

[53] W. Wang, J. Yang and R. Muntz, "PK-tree: A spatial index structure for high dimensional point data," in *Information Organization and Databases*Anonymous Springer, 2000, pp. 281-293.

[54] A. Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching.* ACM, 1984*14(2)*.

[55] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, 1990, pp. 322-331.

[56] T. K. Sellis, N. Roussopoulos and C. Faloutsos, "The R -tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th International Conference on very Large Data Bases*, 1987, pp. 507-518.

[57] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen and D. Šaulys, "Trees or grids?: Indexing moving objects in main memory," in *Proceedings of the 17th ACM*

SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2009, pp. 236-245.

[58] M. Olma, F. Tauheed, T. Heinis and A. Ailamaki, "BLOCK: Efficient execution of spatial range queries in main-memory," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 2017, pp. 15.

[59] S. T. Leutenegger, M. A. Lopez and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *Proceedings 13th International Conference on Data Engineering*, 1997, pp. 497-506.

[60] H. Vo, A. Aji and F. Wang, "SATO: A spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2014, pp. 545-548.

[61] J. Yu, J. Wu and M. Sarwat, "A demonstration of GeoSpark: A cluster computing framework for processing big spatial data," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 1410-1413.

[62] A. S. Abdelhamid, M. Tang, A. M. Aly, A. R. Mahmood, T. Qadah, W. G. Aref and S. Basalamah, "Cruncher: Distributed in-memory processing for location-based services," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 1406-1409.

[63] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 1352-1363.

[64] A. Eldawy, L. Alarabi and M. F. Mokbel, "Spatial partitioning techniques in SpatialHadoop," *Proceedings of the VLDB Endowment*, vol. 8, *(12)*, pp. 1602-1605, 2015.

[65] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, *(11)*, pp. 1009-1020, 2013.

[66] S. You, J. Zhang and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015, pp. 34-41.

[67] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy and T. Qadah, "AQWA: adaptive query workload aware partitioning of big spatial data," *Proceedings of the VLDB Endowment*, vol. 8, *(13)*, pp. 2062-2073, 2015.

[68] C. R. De Rolt, R. Montanari, M. L. Brocardo, L. Foschini and J. da Silva Dias, "COLLEGA middleware for the management of participatory mobile health communities," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, 2016, pp. 999-1005.

[69] M. Lom, O. Pribyl and M. Svitek, "Industry 4.0 as a part of smart cities," in *2016 Smart Cities Symposium Prague (SCSP)*, 2016, pp. 1-6.

[70] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun ACM*, vol. 51, *(1)*, pp. 107-113, 2008.

[71] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, *(13)*, pp. 1565-1568, 2016.

[72] F. Wang, A. Aji and H. Vo, "High performance spatial queries for spatial big data: from medical imaging to GIS," *Sigspatial Special*, vol. 6, *(3)*, pp. 11-18, 2015.

[73] B. Dai and I. Lin, "Efficient map/reduce-based dbscan algorithm with optimized data partition," in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 59-66.

[74] I. M. Aljawarneh, P. Bellavista, A. Corradi, R. Montanari, L. Foschini and A. Zanotti, "Efficient spark-based framework for big geospatial data query processing and analysis," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 851-856.

[75] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, R. Montanari and A. Zanotti, "In-memory spatial-aware framework for processing proximity-alike queries in big spatial data," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD),* 2018, pp. 1-6.

[76] F. Cao, M. Estert, W. Qian and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proceedings of the 2006 SIAM International Conference on Data Mining,* 2006, pp. 328-339.

[77] G. Cardone, A. Corradi, L. Foschini and R. Ianniello, "Participact: A large-scale crowdsensing platform," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, *(1),* pp. 21-32, 2015.

[78] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, 2011.

[79] A. Aji, F. Wang and J. H. Saltz, "Towards building a high performance spatial query system for large scale medical imaging data," in *Proceedings of the 20th International Conference on Advances in Geographic Information Systems,* 2012, pp. 309-318.

[80] V. Mateljan, D. Cisic and D. Ogrizovic, "Cloud database-as-a-service (DaaS)-ROI," in *The 33rd International Convention MIPRO,* 2010, pp. 1185-1188.

[81] S. Cho, S. Hong and C. Lee, "ORANGE: Spatial big data analysis platform," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 3963-3965.

[82] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems.* Springer Science & Business Media, 2011.

[83] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini and R. Montanari, "Efficient QoS-Aware Spatial Join Processing for NoSQL Scalable Storage Frameworks," 2020.

[84] K. Zheng, D. Gu, F. Fang, M. Zhang, K. Zheng and Q. Li, "Data storage optimization strategy in distributed column-oriented database by considering spatial adjacency," *Cluster Computing*, vol. 20, *(4)*, pp. 2833-2844, 2017.

[85] D. Han and E. Stroulia, "Hgrid: A data model for large geospatial data sets in hbase," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 910-917.

[86] Z. Weixin, Y. Zhe, W. Lin, W. Feilong and C. Chengqi, "The non-sql spatial data management model in big data time," in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2015, pp. 4506-4509.

[87] C. Cheng, X. Tong, B. Chen and W. Zhai, "A subdivision method to unify the existing latitude and longitude grids," *ISPRS International Journal of Geo-Information*, vol. 5, *(9)*, pp. 161, 2016.

[88] J. Graça and S. JNdOe, "GeoSharding: Optimization of data partitioning in sharded georeferenced databases," 2016.

[89] K. Li and G. Li, "Approximate query processing: what is new and where to go?" *Data Science and Engineering*, vol. 3, *(4)*, pp. 379-397, 2018.

[90] S. L. Lohr, *Sampling: Design and Analysis*. Nelson Education, 2009.

[91] S. K. Thompson, *Sampling*. Wiley, 2012.

[92] L. Wang, R. Christensen, F. Li and K. Yi, "Spatial online sampling and aggregation," *Proceedings of the VLDB Endowment*, vol. 9, *(3)*, pp. 84-95, 2015.

[93] S. K. Thompson, "Spatial sampling," *Precision Agriculture: Spatial and Temporal Variability of Environmental Quality, (210)*, pp. 161, 1997.

[94] J. Wang, R. Haining and Z. Cao, "Sample surveying to estimate the mean of a heterogeneous surface: reducing the error variance through zoning," *Int. J. Geogr. Inf. Sci.*, vol. 24, *(4)*, pp. 523-543, 2010.

[95] J. Wang, G. Christakos and M. Hu, "Modeling spatial means of surfaces with stratified nonhomogeneity," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, *(12)*, pp. 4167-4174, 2009.

[96] P. Lorkowski and T. Brinkhoff, "Towards real-time processing of massive spatio-temporally distributed sensor data: A sequential strategy based on kriging," in *Agile 2015*Anonymous Springer, 2015, pp. 145-163.

[97] M. Katzfuss and N. Cressie, "Tutorial on fixed rank kriging (FRK) of CO2 data," *Department of Statistics, the Ohio State University, Columbus*, 2011.

[98] J. Wang, A. Stein, B. Gao and Y. Ge, "A review of spatial sampling," *Spatial Statistics*, vol. 2, pp. 1-14, 2012.

[99] D. L. Stevens Jr and A. R. Olsen, "Spatially balanced sampling of natural resources," *Journal of the American Statistical Association*, vol. 99, *(465)*, pp. 262-278, 2004.

[100] A. Grafström, N. L. Lundström and L. Schelin, "Spatially balanced sampling through the pivotal method," *Biometrics*, vol. 68, *(2)*, pp. 514-520, 2012.

[101] I. M. Al Jawarneh, P. Bellavista, L. Foschini and R. Montanari, "Spatial-aware approximate big data stream processing," in *IEEE Global Communications Conference, GLOBECOM*, 2020. To appear.

[102] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, R. Montanari and A. Zanotti, "In-memory spatial-aware framework for processing proximity-alike queries in big spatial data," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2018, pp. 1-6.

[103] A. Lehman, N. O'Rourke, L. Hatcher and E. Stepanski, *JMP for Basic Univariate and Multivariate Statistics: Methods for Researchers and Social Scientists.* Sas Institute, 2013.

[104] A. J. Lister and C. T. Scott, "Use of space-filling curves to select sample locations in natural resource monitoring studies," *Environ. Monit. Assess.*, vol. 149, *(1-4)*, pp. 71-80, 2009.

[105] A. R. Olsen, "Generalized Random Tessellation Stratified (GRTS) Spatially-balanced Survey Designs for Aquatic Resources." *US Environmental Protection Agency, National Health and Environmental Effects Research Laboratory,* 2005.

[106] D. L. Stevens Jr and A. R. Olsen, "Spatially balanced sampling of natural resources," *Journal of the American Statistical Association,* vol. 99, *(465),* pp. 262-278, 2004.

[107] C. V. Networking, "Cisco global cloud index: Forecast and methodology, 2015-2020. white paper," *Cisco Public, San Jose,* 2016.

[108] I. M. Aljawarneh, P. Bellavista, A. Corradi, R. Montanari, L. Foschini and A. Zanotti, "Efficient spark-based framework for big geospatial data query processing and analysis," in *2017 IEEE Symposium on Computers and Communications (ISCC),* 2017, pp. 851-856.

[109] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, R. Montanari and A. Zanotti, "In-memory spatial-aware framework for processing proximity-alike queries in big spatial data," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD),* 2018, pp. 1-6.

[110] S. Tang, Y. Yu, R. Zimmermann and S. Obana, "Efficient geo-fencing via hybrid hashing: a combination of bucket selection and in-bucket binary search," *ACM Transactions on Spatial Algorithms and Systems (TSAS),* vol. 1, *(2),* pp. 5, 2015.

[111] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," *ACM SIGMOD Record,* vol. 28, *(2),* pp. 287-298, 1999.

[112] F. Li, B. Wu, K. Yi and Z. Zhao, "Wander join: Online aggregation via random walks," in *Proceedings of the 2016 International Conference on Management of Data,* 2016, pp. 615-629.

[113] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing,* 2012, pp. 7.

[114] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *ACM SIGARCH Computer Architecture News*, 2014, pp. 127-144.

[115] J. Xie and J. Yang, "A survey of join processing in data streams," in *Data Streams*Anonymous Springer, 2007, pp. 209-236.

[116] R. T. Whitman, M. B. Park, B. G. Marsh and E. G. Hoel, "Spatio-temporal join on apache spark," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2017, pp. 20.

[117] M. A. Naeem, O. Aziz and N. Jamil, "Optimising HYBRIDJOIN to Process Semi-Stream Data in Near-real-time Data Warehousing," 2019.

[118] R. Derakhshan, A. Sattar and B. Stantic, "A new operator for efficient stream-relation join processing in data streaming engines," in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, 2013, pp. 793-798.

[119] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys (CSUR)*, vol. 24, *(1)*, pp. 63-113, 1992.

[120] M. A. Naeem, K. T. Nguyen and G. Weber, "A multi-way semi-stream join for a near-real-time data warehouse," in *Australasian Database Conference*, 2017, pp. 59-70.

[121] N. R. Herbst, S. Kounev and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, 2013, pp. 23-27.

[122] T. Lorido-Botrán, J. Miguel-Alonso and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech.Rep.EHU-KAT-IK-09*, vol. 12, pp. 2012, 2012.

[123] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava and J. Widom, "Stream: The stanford data stream management system," in *Data Stream Management*Anonymous Springer, 2016, pp. 317-336.

[124] T. Heinze, Z. Jerzak, G. Hackenbroich and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014, pp. 13-22.

[125] L. A. Combaneyre, "Minority Report is Here—Real-Time Geofencing Using SAS® Event Stream Processing," *Paper SAS395-2017, Available Online at: Http://Support.Sas.Com/Resources/Papers/proceedings17/SAS0395-2017.Pdf*, pp. 1-10, 2017.

[126] X. Chen, Y. Vigfusson, D. M. Blough, F. Zheng, K. Wu and L. Hu, "GOVERNOR: Smoother stream processing through smarter backpressure," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 145-154.

[127] T. Heinze, V. Pappalardo, Z. Jerzak and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, pp. 296-302.

[128] J. M. Hellerstein, P. J. Haas and H. J. Wang, "Online aggregation," in *Acm Sigmod Record*, 1997, pp. 171-182.

[129] P. Carbone, A. Katsifodimos and S. Haridi, "Stream Window Aggregation Semantics and Optimisation," *Stream Window Aggregation Semantics and Optimization.*, 2019.

[130] R. A. Sugden, T. Smith and R. P. Jones, "Cochran's rule for simple random sampling," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 62, *(4)*, pp. 787-793, 2000.

[131] 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.

[132] J. Fang, R. Zhang, X. Wang and A. Zhou, "Distributed stream join under workload variance," *World Wide Web*, vol. 20, *(5)*, pp. 1089-1110, 2017.

[133] S. You, J. Zhang and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015, pp. 34-41.

[134] H. Sun, R. Birke, W. Binder, M. Björkqvist and L. Y. Chen, "AccStream: Accuracy-aware overload management for stream processing systems," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 39-48.

[135] A. I. Maarala, M. Rautiainen, M. Salmi, S. Pirttikangas and J. Riekki, "Low latency analytics for streaming traffic data with apache spark," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 2855-2858.

[136] F. L. Hall, "Traffic stream characteristics," *Traffic Flow Theory.US Federal Highway Administration*, vol. 36, 1996.

[137] P. Mehta, A. Voisard and S. Müller, "Clustering spatial data streams for targeted alerting in disaster response," in *Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming*, 2013, pp. 66-75.

[138] C. C. Aggarwal, J. Han, J. Wang and P. S. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th International Conference on very Large Data Bases-Volume 29*, 2003, pp. 81-92.

[139] C. C. Aggarwal, *A Survey of Stream Clustering Algorithms.* Chapman and Hall/CRC, 2018.

[140] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R -Tree: A Dynamic Index for Multi-Dimensional Objects." 1987.

fin.