Thesis for The Degree of Doctor of Philosophy

Enhancing Localization, Selection, and Processing of Data in Vehicular Cyber-Physical Systems

Bastian Havers-Zulka

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2024

Enhancing Localization, Selection, and Processing of Data in Vehicular Cyber-Physical Systems

Bastian Havers-Zulka

ISBN 978-91-8103-002-0
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5460.
ISSN 0346-718X

Department of Computer Science and Engineering
Division of Computer and Network Systems
Chalmers University of Technology
SE-412 96 Gothenburg,
Sweden
Phone: +46(0)31 772 1000

Cover image:

A modern vehicle, equipped with an on-board computer that stores and processes data from a visual sensor, connected to a remote server *(vehicle artwork generated with DALL·E 3)*.

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2024.

"But need alone is not enough to set power free: there must be knowledge."
- Ursula K. Le Guin

Enhancing Localization, Selection, and Processing of Data in Vehicular Cyber-Physical Systems

Bastian Havers-Zulka

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Connected devices on the edge of the *Edge-to-Cloud* (E2C) continuum are producing increasing amounts of data that hold the key to unlocking valuable use cases among a wide range of applications. In the vehicular domain, connected vehicles in large fleets (called *Vehicular Cyber-Physical Systems* or VCPSs) sense and collect terabytes of data such as time series and video, enabling everything from predictive maintenance to autonomous drive. For VCPSs the computing devices located onboard vehicles are not dimensioned to process all the data produced onboard. Simultaneously, communication to the cloud, where computing resources are more readily available, relies on bandwidth-limited and costly carrier-operated cellular connectivity. As transmitting all raw data to the cloud for analysis incurs increasing costs and processing latencies, and the edge devices lack the capability to perform all required data analyses, the questions of *where* and *how* to process *which* data become paramount and form the foundation of this thesis. The first part of this thesis gives an outline of my work by introducing relevant background topics, motivating the research questions and describing the contributions of this thesis. These contributions are then contained in the five chapters that make up the second part: in Chapter A, I present the DRIVEN framework consisting of a novel lossy online time-series compression algorithm with tuneable bounded error for the edge, as part of a pipeline from edge to cloud that includes online data clustering, and evaluate the tradeoffs between data savings and reduced analysis accuracy from lossy compression. In Chapter B, I show how our work on Data Localization helps in discovering those vehicles in a connected fleet that have data relevant to a user-defined analysis task quickly and efficiently. Chapter C proposes Ananke, the first forward provenance framework for Stream Processing, enabling a route for selecting relevant data inside streaming sources that are ubiquitous in VCPSs. In Chapter D, I present the Nona framework that solves the problem of forward provenance for evolving sets of Stream Processing queries and thus allows data selection for modern analysis flows in which queries are constantly altered and redeployed. Finally, in Chapter E, I introduce a comprehensive requirements list for and an implementation of a VCPS learning simulator that enables the efficient evaluation of distributed data analysis algorithms for connected vehicular networks. This thesis makes significant steps forward for utilizing edge resources more efficiently, while also setting the basis for further development of novel distributed data analysis algorithms in VCPSs.

Keywords: Stream Processing, Edge-to-Cloud Continuum, Distributed Data Analysis, Provenance, Vehicular Cyber-Physical Systems

Acknowledgments

Thank you,
Vincenzo, for taking me onto this six-year journey, for having my back, engaging in fruitful discussions on and off-topic, belaying me literally and figuratively, and an always open door; Marina, for all the guidance and motivation you provided, and your ability to put things into and see things from the right perspective; Romaric, for all fruitful collaborations in the office and the climbing gym; Dimitris and Hannah, for doing fantastic work together and creating a great atmosphere in the offices; and Philippas, for providing wisdom and examining my PhD studies.
Thank you,
everyone I've had the pleasure to work with at Volvo Cars; Peter Härslätt, Ashok Koppisetty, Max Petersson, Ivana Hrvoj, Hampus Grimmemyhr and Erik Hjerpe for enabling, being a part of, and believing in my research and my project; and Asli, Herman, Robin, and Koen, and all my former and present team members.
Thank you,
all the people and friends I've met at Chalmers; Christos, Fazeleh, Georgia, Thomas, Kalle (also for interesting discussions), Kåre, Martin, Oliver, and Valentin, for mastering our PhDs together, in and outside the office; and Ahmed, Aljoscha, Amir, Aras, Babis, Carlo, Elad, Erik, Francisco, Huaifeng, Ivan, Jacob, Jingyu, Joris, Kim, Magnus, Mohamed, Nasser, Olaf, Philippas, Tomas, Vinh, Wania, and many more.
Thank you,
dear parents and dear sisters, for supporting me throughout my life, for helping me get to where I am now, and for always believing that I will not throw in the towel before reaching the finish line; and dear friends in Gothenburg, including Anshuman, Alina, Birte, David, Érika, and Steffi, for helping me keep my spirits high; and to all my friends in Germany, including Paul and Lea, who helped make Sweden feel closer.
And, most of all, thank you,
Linn, my wife and friend, for your unwavering love and support in my life; without you, I would not even have started this thesis; and Minna, the most fantastic daughter one could ever wish to have, for filling every day with joy.

Funding Sources: I wish to acknowledge financial support by VINNOVA, the Swedish Government Agency for Innovation Systems, projs. "Onboard-/Offboard Distributed Data Analytics (OODIDA)" (DNR 2016-04260) and "Automotive Stream Processing and Distributed Analytics (AutoSPADA) / OODIDA Phase 2" (DNR 2019-05884) in the funding program FFI: Strategic Vehicle Research and Innovation.

Research Papers and Contributions

Appended Research Papers

This thesis is based on the following research papers (the capital index letter refers to the corresponding chapter of this thesis):

[A] **Havers, B.**, Duvignau, R., Najdataei, H., Gulisano, V., Papatriantafilou, M., and Koppisetty, A.C., "DRIVEN: A Framework for Efficient Data Retrieval and Clustering in Vehicular Networks", *Future Generation Computer Systems. Vol. 107, p. 1-17 (2020).*

[B] Duvignau, R., **Havers, B.**, Gulisano, V., and Papatriantafilou, M., "Time- and Computation-Efficient Data Localization at Vehicular Networks' Edge", *IEEE Access. Vol. 9, p. 137714-137732 (2021).*

[C] Palyvos-Giannas, D., **Havers, B.***, Papatriantafilou, M., and Gulisano, V., "Ananke: A Streaming Framework for Live Forward Provenance", *Proceedings of the VLDB Endowment. Vol. 14 (3), p. 391-403 (2020).*

[D] **Havers, B.**, Papatriantafilou, M., and Gulisano, V., "Nona: A Framework for Elastic Stream Provenance", *under submission (2024).*

[E] **Havers, B.**, Papatriantafilou, M., Koppisetty, A.C., and Gulisano, V., "Proposing a Framework for Evaluating Learning Strategies in Vehicular CPSs", *Proceedings of the 23rd International Middleware Conference Industrial Track, Part of Middleware 2022, p. 22-28 (2022).*

*The first two authors contributed equally to this publication.

Other Research Papers

The following research papers, while published during my PhD studies, are not appended to this thesis due to contents that overlap with those of appended publications or that are complimentary to the thesis.

Havers, B., Duvignau, R., Najdataei H., Gulisano, V., Koppisetty, A.C., and Papatriantafilou, M., "DRIVEN: A Framework for Efficient Data Retrieval and Clustering in Vehicular Networks", *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE), p. 1850-1861 (2019).*

Duvignau, R., **Havers, B.**, Gulisano, V., and Papatriantafilou, M., "Querying Large Vehicular Networks: How to Balance On-Board Workload and Queries Response Time?", *Proceedings of the IEEE Intelligent Transportation Systems Conference (ITSC) 2019, p. 2604-2611 (2019).*

Gulisano, V., Palyvos-Giannas, D., **Havers, B.**, and Papatriantafilou, M., "The Role of Event-Time Order in Data Streaming Analysis", *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, p. 214-217 (2020).*

Havers-Zulka, B., "Distributed and Communication-Efficient Continuous Data Processing in Vehicular Cyber-Physical Systems", *Licentiate Thesis (2020).*

Personal Contributions

I contributed to Chapter A, Chapter D and Chapter E as lead designer and main implementer throughout, with the exception of the clustering algorithm *Lisco* adapted and extended in Chapter A, which was developed in collaboration with all other authors. I prepared and performed the evaluations and was the chief responsible for writing the manuscripts in these three publications. In Chapter C, Dimitris Palyvos-Giannas and I shared all roles equally. In Chapter B, I was the chief responsible for the evaluations and co-responsible in the design and implementation led by Romaric Duvignau. Romaric Duvignau and I contributed equally to the writing of that manuscript.

Contents

Part I

Thesis Overview

1 Introduction

The speed of light and data movement energy will limit what cloud-only solutions can deliver.
- Ada Gavrilovska [140]

A large share of today's data growth occurs along the *Edge-to-Cloud* (E2C) continuum [140] that encompasses networked devices in hierarchies from low-powered and embedded edge devices up to powerful and scalable cloud servers. This growth in data produced by sensors and recorded by heterogeneous devices is rapid: projections suggest exponential growth of the datasphere, with a doubling of data output roughly every three years [167] - from about one zettabyte (= one billion terabytes) in 2010 to 200 zettabytes in 2025 [184]. To illustrate this, the information from 200 zettabytes, printed out to book pages, suffices to cover the entire surface area of the world, including water, in a stack of paper about 30 cm high[(a)]. This surge in information led to the the term *Big Data* to characterize the vast, constantly growing datasets whose processing marks some of the greatest computational challenges today.

Big Data is already allowing for groundbreaking insights and developments across diverse domains, from analyzing stock market shifts [14] to predicting the wearing down of a car's brake pads [103], and from training AI assistants [26] to progresses in autonomous mobility ([68], [206]). Due to its immense potential, Big Data has been likened to a new economic asset, often referred to as "the new oil" or as the equivalent of raw gold [200]. Big Data is invaluable for offering up unforeseen insights through exposing hidden correlations. The sheer size of modern data sets plays a pivotal role for the development of novel Machine Learning tools such as image recognition [115] and recent Large Language Models [26] that mark the advances of digital automation.

Similar to oil and raw gold, where maximum value is unlocked through extraction and refining or mining and goldsmithing, Big Data has to be aggregated, moved, and analyzed to eventually lead to insights that help solve problems. Just as the demand for petroleum products has encouraged the development of novel machinery that facilitates the various extraction and processing steps of crude oil, the interest in data products has led to the development of new processing techniques and paradigms that can cope with Big Data. However, as Big Data continues to grow in size and velocity, existing data processing techniques and paradigms must be further refined, and the demand for innovations continues. The following trends exemplify those issues, and present further avenues for development:

First, while data growth follows an exponential path, the average mobile data speed is recently growing only linearly [183]. While new technologies such as 5G are being introduced and next-generation technologies researched [105], it requires time and large investments for novel cellular standards to find a wider adoption. Simultaneously, data is increasingly sensed at the edge of networked

[(a)] Assuming 300 words per page and a page thickness of 0.1 mm.

systems, for example by low-powered sensors in smart factories or measuring devices distributed throughout modern vehicles. Transmitting this data to powerful cloud servers for analysis strains the communication infrastructure and introduces additional latencies into the data processing from sending large data over networks not dimensioned for this purpose. One approach to avoiding bottlenecks created from the growing chasm between network bandwidth and data volumes, especially for data produced towards the far ends of the E2C continuum where wired network access becomes impossible, is to *process data close to where it is being produced*. This is summed up by Ada Gavrilovska's quote in the beginning of this section: while computing hardware in the cloud is powerful and scalable, reliance on the cloud alone increases latencies and strains the communication infrastructure as data has to be transported from the edge inwards.

Second, continuing with the processing difficulties faced in the E2C continuum, in 2020, it was estimated that the amount of data produced already exceeded the worldwide storage capacity by a factor of ten [184]. While from 2010 to 2020, the data output increased by a factor of thirty, in the same time span the cost of disk storage went down only by a factor of three [137]. Data growth is already outpacing the decreases in storage cost, and at current trajectories that gap is poised to widen. As a consequence, one must be aware that *not all raw data can be stored before it is processed*, especially given that a large share of the data is produced on edge devices that are low-powered, numerous, and thus especially cost-sensitive.

Combining the trends of data growth versus mobile speeds, and storage availability and costs especially in the edge domain, an additional aspect emerges: Maybe *not all data should be treated equally*. While it is generally not possible to judge which data is important for a certain analysis without having seen the data, smart and efficient pre-emptive filtering and selection can ease the stress both on communication networks and storage and processing facilities. That way, only selected data becomes subjected to costly operations such as transmission, storage or full in-depth analysis, while less relevant or possibly redundant data is weeded out early.

The above discussions, while relevant for numerous systems that can be characterized under the E2C continuum, are especially significant for *Vehicular Cyber-Physical Systems (VCPSs)* - Systems of connected vehicles, equipped with onboard sensors and computing units, that can communicate wirelessly with the cloud. In this thesis, I will discuss novel techniques that help alleviate communication and processing bottlenecks by *distributing* computation in VCPSs and leveraging *Stream Processing*, a continuous and iterative processing paradigm that does not require data to be stored before it is processed. Note that, while this thesis highlights VCPSs, the presented ideas and techniques are equally relevant for other systems in the E2C continuum.

Organization This thesis consists of two parts. The first one, beginning with this Introduction, continues in Section 2 with a more detailed presentation of VCPSs. Section 3 then overviews preliminaries relevant for approaching data processing near the edge; Section 4 presents the research methodology I

applied; Section 5 motivates the research questions that structure and guide the work; Section 6 overviews and discusses the contributions of this thesis; and Section 7 concludes with a summary and an outlook.

The second part of this thesis, Part II, contains the detailed thesis contributions in five chapters, as disseminated in the publications resulting from my PhD studies.

2 Vehicular Cyber-Physical Systems (VCPSs)

Due to the large value inherent to the data produced in them, and the special challenges arising, for example, from the mobility of the system's actors, VCPSs are of special interest for this thesis. In the following, I will present their defining characteristics, before framing the key challenges for data processing in such systems against the backdrop of VCPSs.

2.1 Defining Characteristics

For decades, cars were equipped only with a small set of analog sensors that operated independently and presented information directly to the driver. Examples of these include mechanical vehicle speed or engine temperature gauges. The sensor output was processed directly by the human in charge: to accelerate or brake depending on the speedometer reading or to let the engine cool should the temperature gauge indicate overheating. It took until the late 1970s for the first *connected* sensors to be installed in cars, together with the first microcontroller-based control unit in the form of an engine control module to optimize the combustion process. This module could take autonomous control decisions based on input from temperature and exhaust flow sensors [70].

The size of a vehicle's sensor set has only grown in the decades since, from around 20 in the early 2000s to more than 100 in vehicles today [69]. Looking at the data amounts created in a modern vehicle, one is facing several gigabytes per hour [44] - an amount of data that in the 1970s could barely fit storage systems the *size of a vehicle*. Already a decade ago, the literature started considering cars as producers of *Automotive* Big Data ([21], [106]). With the ongoing introduction of advanced semi-autonomous driver assistance technologies, the industry is seeing the addition of even higher-bandwidth sensors such as Radar, multiple stereo and 3D cameras, and LiDARs (an array of laser-based distance sensors that create a 3D map of the surroundings) to the already complex sensor set of vehicles. As an example, a Volvo EX90 will include, in addition to dozens of traditional lower-bandwidth sensors, eight cameras, five radars, and a LiDAR sensor generating more than two million data points per second [133]. The data amounts produced by these novel sensors are orders of magnitude larger than those of tire-pressure sensors and simple GPS beacons [203] used in traditional vehicles, leading to a drastic increase in data produced per vehicle.

With increasing demands for active safety functionality (such as emergency breaking or driver monitoring for drowsiness detection) and convenience driver assist functions (such as highway pilots and parking assistants), cars have also been equipped with more powerful computing hardware. Modern hardware [145] can fuse heterogeneous sensor data and execute complex assignments such as running inference for computer vision models. In spite of this, cars are *very resource-constrained systems*: the computing hardware is tailored precisely to cater to the needs of safety- and general driving-critical functionalities to reduce costs that are multiplied over entire vehicle fleets. Furthermore, CPU and GPU power draws can become concerning when competing for range among battery-electric vehicles.

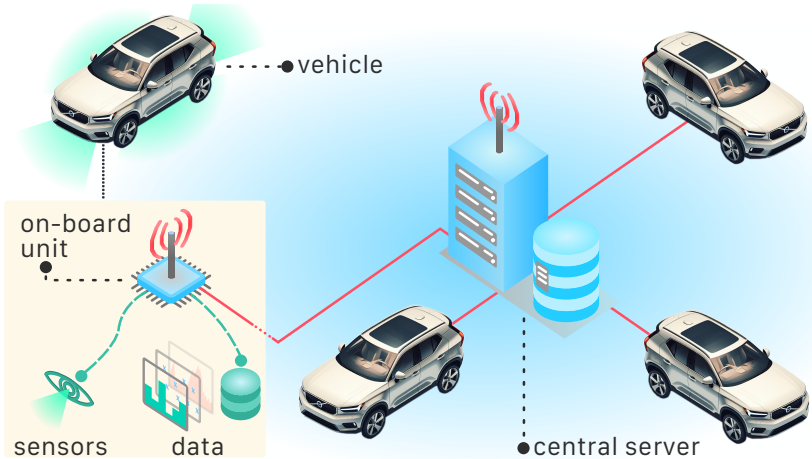


Figure 1: VCPS: A wirelessly connected vehicle fleet. As the detailed view on the left shows, each vehicle possesses an on-board unit that can process data generated by the vehicle's sensors. The on-board units effectively form the *edge* of the network. At the central server, analyses over the data from the edge can be coordinated.

Moreover, it is important to note that while Automotive Big Data from individual cars can be important to provide individualized analysis and functionality, the larger interest is in data from larger fleets that span tens of thousands of vehicles. This is where Automotive Big Data becomes truly big and valuable. Some examples of applications include online congestion monitoring in *smart* cities, the development of novel applications such as hazard warning systems [32] and platooning (vehicles autonomously following each other close enough to minimize air drag for follower vehicles [13]), diagnostic applications such as Predictive Maintenance (e.g., [160]), driver interaction analysis to cater to new driver requirements and preferences ([73], [175]), and of course the development of Autonomous Driving. Estimated to be worth some *1.5 Trillion USD* by 2030, car data and corresponding applications are on track to become a main ground of operations for vehicle manufacturers and fleet owners [138].

As vehicles are increasingly connected to the internet (and, in some capacity, directly to each other; [44], [162]), an entire fleet of vehicles can be seen as a single connected system - a VCPS, with cars sensing data and having some limited computational headroom on an *on-board computer* able to process and store sensor data. Via their internet connectivity, cars can communicate to central servers. These can be seen as an abstraction for rented capabilities in the cloud, or on-premise servers at vehicle fleet owners. The sensors and on-board computers are located at the outskirts of the system, physically separated from the central server, and constitute the *edge*. Figure 1 shows a sketch of such a VCPS. As an example application, an analyst in Figure 1 at the central server could request to transmit LiDAR point cloud data from the edge to the server for further processing (an example use case from Chapter A),

or they could task the vehicles to report whether they have passed a point of interest in the last day (from Chapter B). In other cases, however, insights may be needed at the edge itself to, for example, employ Machine Learning models trained on data *from* the vehicles *on* the vehicles themselves (see Chapter E).

2.2 Key Challenges for Data Processing

In the following, I will describe in more detail two of the defining characteristics of a VCPS, *connectivity* and *processing capabilities*, to highlight the resulting challenges for data processing.

Connectivity Almost every vehicle sold today is equipped with a cellular antenna to communicate via carrier-operated networks with the cloud [162]. This connection relies on 3G, 4G or 5G wireless technology. As mentioned in Section 1, even the fastest cellular connectivity available is limited compared to the amount of data that can be sensed at the edge. An additional factor limiting the amount of data that can be transmitted is the monetary cost of using such carrier-operated networks.

It should be noted that further communication technologies for vehicles exist. Chiefly, these are V2V, used to describe a short-range vehicle-to-vehicle link, and cellular V2X [100], which incorporates short- and long-range communication with other vehicles, road-side units, or cellular networks. V2V or V2X communication can avoid carrier-operated networks and thus offer interesting possibilities for reducing communication costs. However, the required short range between vehicles poses challenges, with an accompanying body of literature (e.g., [144]). According to an industry study [43], V2V and V2X are not yet widely rolled out[(b)]. Unless otherwise stated, I focus in my work on the use of a direct cellular link for VCPSs[(c)].

Processing capabilities To process the data from more than 100 sensors and perform operations such as emergency braking or lane-keeping, modern vehicles possess special-purpose on-board computers whose capabilities vary depending on their specific use case. However, novel semi- and fully autonomous vehicles (with especially the latter still being in the experimental or concept phase) require the real-time processing of large amounts of camera, Radar and LiDAR data and are usually equipped with a central (and possibly redundantly implemented) computing unit comparable to or exceeding modern consumer hardware (an example is the DRIVE platform by chip manufacturer NVIDIA [145]). With power draws in the range of 200 to 2000 W [71], these computing units compete with other components of the vehicle for the total

[(b)] In the US, for example, the promise of V2V was considered unfulfilled and parts of the allocated communication spectrum had to be returned [11].

[(c)] Exact details of the used communication protocols are orthogonal to the techniques presented in this thesis and abstracted in my work. Thus, the routing of messages through the network is not covered in detail. Sophisticated routing solutions in V2X networks could allow the delivery of messages from the cloud to individual vehicles similar to direct cellular links; likewise, vehicles can talk to other vehicles indirectly using cellular connectivity.

available electric power, an effect that is exacerbated in battery-electric vehicles in which battery capacity further limits the power headroom. Consequentially, additional and spontaneous analyses deployed on the on-board computer can only be allowed to have a relatively small impact. This is in contrast to the central server in a VCPS, which can be situated in the cloud to be easily scalable, and can be assumed to be dedicated to performing data analysis, both in terms of computation and storage.

With connectivity bottlenecks and limited, but distributed processing capability on the one hand, and the large interest in processing Automotive Big Data on the other, three key questions arise:

Key Questions for Data Processing in VCPSs

1. **Where** in a VCPS should data be processed?
2. **How** should data be processed?
3. **Which** data should be prioritized?

Where in a VCPS should data be processed? Focusing first on the *Where?*, we observe the following: While data is sensed at the system's edge, where typically for low-powered devices available processing power is limited, the powerful central server can leverage orders of magnitude more resources. Thus, it can appear that processing the data from one or several vehicles centrally is a good solution to the cars' being resource-constrained in their computational capabilities. However, as raised in [40], considering the limitations imposed by the wireless network, transmitting raw data from the vehicles to the central server quickly becomes infeasible for analyses that require large amounts of data from many vehicles. In addition to insufficiencies of the network, which cannot sustain transmitting the required data amounts for analysis of VCPS data (as also highlighted in [28]), the costs for moving this much data over carrier-operated networks would be prohibitive. Moreover, with automotive fleets in the range of 100,000s of vehicles, processing and storing the raw data from all vehicles can still exceed the central processing and data management infrastructure [106].

On the other end of the spectrum of distributing the analysis in a VCPS, all computation is done on the vehicle's computing units, and the central server only serves to initiate the data processing. Thereby, costs and bottlenecks imposed by data transmission can be almost completely avoided. However, the demanded analysis may exceed the available local computing budget of an on-board device (as noted in, e.g., [130], [209]), or the analysis may need to span the data from several vehicles and thus be impossible when done in a completely siloed fashion. As recognized in [136], efficient approaches to pushing calculations to the edge of the E2C continuum in a VCPS need to adapt the respective analysis pipeline in order to utilize the available computational power in the complete network (central server *and* edge nodes), while minimizing the transferred data amounts.

How should data be processed? The answer to the question of *How?* is strongly coupled to the use case at hand. However, any approach that involves storing large amounts of data on the vehicle before they are processed on-board or sent to the cloud inherently creates additional latency for the analysis, and may require large storage facilities on-board because of the vast data amounts produced. Transmitting raw data immediately after its generation from the vehicle to the cloud only to store it there before processing has the same drawbacks for time-sensitive analysis. Furthermore, while storage appears less constrained in the cloud, storing the data from large fleets of vehicles can still prove to be costly.

To forego the shortcomings of a storage-intensive approach, *Stream Processing* emerges as a strong candidate paradigm for data processing in low-power and storage-constrained systems. Instead of traditional data processing that first gathers all data into one place and then performs a global analysis, Stream Processing assumes that there is no end to the data, and that new data will continuously be produced. With this assumption, Stream Processing can continuously analyze data and does not need to wait for some global aggregation of all data first, reducing the amount of storage needed. While it does allow analyses that span large amounts of data at once, it also covers those use cases where results are required continuously and as quick as possible. Additionally, Stream Processing scales well with the available hardware and thus lends itself favorably to resource-constrained environments encountered on the edge.

Which data should be prioritized? Performing costly operations (such as data transmission and extensive analysis) only on data relevant for a use case at hand can alleviate pressure from the system bottlenecks, freeing up processing capabilities and bandwidth from the edge to the center. As an example, by identifying those vehicles in the fleet that have recorded relevant data, it is possible to instruct only these identified vehicles to transmit their data to the central server for centralized analysis, or to push such an analysis only to them (if such an identification can be done quickly and computationally cheaply).

On the level of streams of data from a set of sensors, relevant data can be identified and marked for further processing, transmission or storage - effectively *selecting* data for further use, and allowing to discard or de-prioritize less relevant pieces of data. In an example adapted from Chapter C, an analyst may want to select only those video frames from a vehicle's front camera for further analysis that show a cyclist stopping in front of the vehicle, and discard all other frames. This example also shows that such selection in many cases must be *stateful*, taking into account not only a single piece of data (a single frame), but several over time (here, to decide whether the cyclist was stopping or simply crossing in front of the vehicle).

In the following chapter, I will present techniques and paradigms that are helpful in approaching these three questions.

3 Background

3.1 Stream Processing

Big Automotive Data is continuous in nature: as vehicles are driven, their sensors continuously generate data, sensing evolving physical phenomena, be it outside the car, in the cabin, or inside the engine or battery pack. This type of data is only somewhat compatible with classical database technologies such as SQL or noSQL, which are built on the predicate of *first-the-data-then-the-query*: data sets are collected, stored, and subsequently queried and processed, which can incur several drawbacks when faced with continuous data. Storing data before it is processed leads to inherent latency, as first a complete data set has to be compiled. Thus, results pertaining to early data may be outdated by the time the complete data set is compiled and the analysis generated. Also, large storage capacities are required to (at least temporarily) store the complete data set. Finally, it may be required, but computationally expensive, to reprocess large chunks of data in this batch fashion once novel raw data arrives.

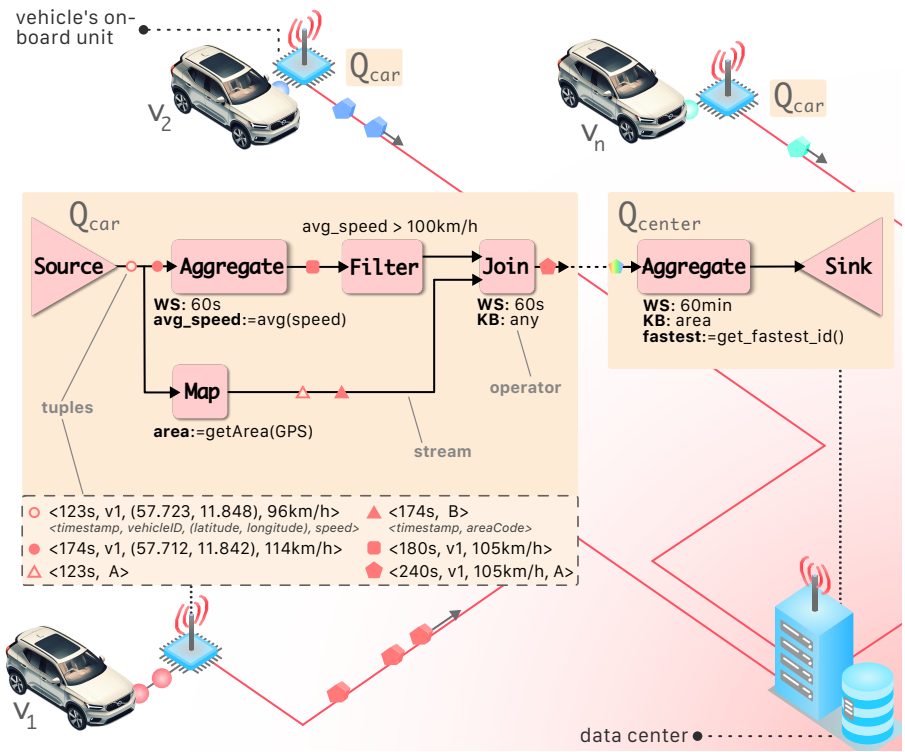


Figure 2: A Stream Processing query that detects the fastest vehicles per area (A/B) every hour. The source produces tuples with the schema ⟨timestamp, ID, (lat,lon), speed⟩. The query is distributed; some stages (Q_{car}) are deployed on-board vehicles and some (Q_{center}) at the central server.

The Stream Processing paradigm [180] represents an alternative to classic database designs: instead of storing data first and deploying *queries* later to generate insights, the query is defined first. Subsequent new data is continuously queried upon entering the processing pipeline, and incremental aggregations ensure that updated results can be obtained without having to reprocess past data. To give a more thorough introduction to Stream Processing, the following paragraphs will detail its key notions.

Streams, Operators and Queries

A Stream Processing *query* is a set of transformations that act on streams of data [85], resulting in a Directed Acyclic Graph (DAG) through which the data streams flow. An example query could be the processing of a stream of positional reports from a fleet of vehicles to continuously find out which is the fastest car in the last hour in a certain geographic area, as shown in Figure 2. Such queries are processed by Stream Processing Engines (SPEs) as Apache Flink [29] or Storm [10]. Data travels in the streams as *tuples*, which commonly are timestamped with regards to the physical time of their creation (the *event time*), for example the time a sensor reading has occurred. Each query begins with one or several *sources*. For example, these can be a single physical sensor that continuously reports sensor readings, such as a GPS receiver or a wheel speed sensor, a group of sensors such as a rotating LiDAR (e.g. a Velodyne HDL-64E which consists of 64 individual laser distance sensors [155]), or an interface to an upstream system such as a database or a message broker service. In Figure 2, the source is a GPS receiver that emits the timestamp, GPS position, and current speed of the vehicle. The tuples produced by sources are called *source tuples*. From the source(s), source tuples flow to *operators*, the semantic units that manipulate and create tuples. Operators may have several input and output streams (in Figure 2, the Join operator has two inputs). Eventually, tuples end up as *sink tuples* at *sinks*, which may be file sinks that write incoming tuples to storage, or interfaces to other downstream processing and storage systems.

Operators can be subdivided into two types, *stateless* and *stateful* operators. Stateless operators act on individual tuples and produce one or more output tuples. Their *statelessness* means that the operator does not keep a state that evolves with the tuples, and thus the possible output tuples do not depend on previously seen tuples. Examples of stateless operators are the *Map* and *Filter* operators, which are also employed in Figure 2 to map from GPS location to geographic area or to filter out tuples with a too low speed reading. Stateful operators, on the other hand, process delineated groups of tuples that typically span finite time periods either relative to the processing time or relative to the tuples's event time. These time periods are called *windows*. Examples of stateful operators include the *Aggregate* and *Join*, also depicted in the figure: The first Aggregate collects all tuples in a 1-minute window and produces one output tuple with the average speed of the aggregated tuples, while the Join matches any pair of tuples from its two input streams within the last minute, producing an output tuple carrying both the average speed and the mainly

visited location. Using keys of the input tuples, parallel instances of stateful operators that each ingest only a partition of tuples can be deployed: as an example, the second Aggregate in the figure produces the ID of the fastest vehicle of the last minute keyed by (KB) area. While these *native* operators are common to most widely used SPEs, many SPEs allow the implementation of user-defined operators that extend the native operators's functionality.

Watermarks

It can in general not be assumed that the timestamps of tuples inside a stream are sorted. For example, tuples may be read from sources that already deliver the data disordered, or they become disordered inside a query because of parallel operator instances. To reliably observe the passage of time in spite of disorder, the concept of *watermarks* [99] has been adopted by many SPEs (e.g., [10], [29]), which provides a partial order on streams and thus bounds disorder. Thereby, watermarks can enable deterministic (and reproducible) processing.

Applicability in VCPSs

Having evolved for distributed, parallel and elastic online analysis (see, e.g. [29]), and scaling from low-powered devices (see, e.g., Chapter A) to manycore systems (Chapter C), the Stream Processing paradigm is favorably employed in E2C systems [46] such as VCPSs. An example of a Stream Processing application deployed in a VCPS is shown in Figure 2. This application is distributed between n vehicles $v_1, \ldots, v_n$, and a central server. Q_{car}, the first part of the query and a subgraph of the application DAG, is deployed in parallel at each vehicle. The last operator of Q_{car} produces tuples that are wirelessly transmitted to the data center, where the second part of the query is deployed, Q_{center}. The DAG of the query is thus purposely split into connected subgraphs, in this case to exploit computational resources on the vehicles and reduce the amount of data sent wirelessly by performing aggregate operations already on the vehicle.

3.2 Time-Series Data Compression

As underlined in Section 2.1, the amount of data generated especially in VCPSs is growing rapidly, thereby stressing the computing and communication infrastructure. At the edge, available storage is typically smaller than at central utilities, and transmission to these may be costly and slow [161]. At central utilities, while storage capacity is larger, usually the data from a large number of edge devices has to be stored. As we have seen, one way to reduce raw data amounts is through filtering, aggregation or prioritization. A widely used additional approach is the compression of raw sensor data, which can alleviate the storage bottlenecks occurring at the edge and at central utilities as well as reduce the cost and duration of data transmission [15], [91].

While the goal of data compression is always to find a smaller representation of data, different families of techniques rely on different paradigms for

data compression, and optimize for different metrics. One such metric is the *faithfulness* of the data after undergoing compression and then decompression (reconstruction). Some techniques exist that achieve compression in a lossless fashion, e.g. Delta Compression [181] for integer values or data-agnostic dictionary compression as used in the `DEFLATE` algorithm of ZIP [52]. Other approaches aim at *approximating* time-series data, for example [33], [61], incurring a loss in precision of the compressed representation. Here, I will focus on lossy compression of numerical multivariate time-series data.

Piecewise-Linear Approximations (PLAs) of Time-Series Data

Numerical multivariate time-series data is a series of timestamped data points describing a $N+1$-dimensional curve, with time being one of the dimensions. By pairing each of the N dimensions with time, it can be decomposed into $2N$ univariate time series, where every time series has just one variable that depends on time. As one example, let us regard the point cloud created from a rotating LiDAR (see [91]). A LiDAR column is a rotating stack of usually 64 laser-based distance sensors that, after one full rotation, create a three-dimensional map of the surroundings through the angle and distance of objects that reflect the laser beams back to the stack. Mapping this to 64 two-dimensional time series, one obtains one time series per laser.

To compress a univariate time series, one approach is to calculate a piecewise-linear function that best approximates the original data (see Figure 3 for an example involving the time series of a single laser from a LiDAR). The set of all linear *segments* of the approximation results in a (typically) smaller encoding than the uncompressed data, although the viability of the method is subject to the underlying data (some techniques are adaptive and can avoid producing a larger representation [58]). Being lossy, a PLA of, for example, a LiDAR point cloud may result in a slightly distorted point cloud due to the approximation error, once decompressed. That may mean that the physical objects that appear in the cloud appear as shifted from their true position, but the required storage space for this point cloud may be drastically reduced by placing points in the cloud along straight lines where possible [91].

Although differing in their approach and optimized for various metrics (e.g., fast search [127]), many of the PLA techniques in the literature have in common that they allow varying the approximation error or maximum

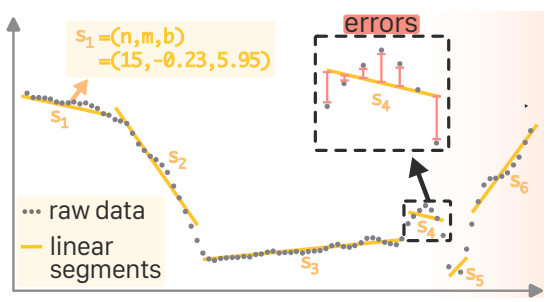


Figure 3: A piecewise-linear approximation of the dotted grey time-series (coming from a LiDAR) via the orange segments s_i. The inset shows the approximation errors between the original and linearly approximated data. Segments are expressed via three parameters: their length, slope, and y-intercept `(n,m,b)`.

deviation between the original data and the piecewise-linear representation. A larger error typically results in longer segments and thus a reduced size of the encoding. Thus, approximation techniques can typically achieve smaller representations than lossless compression algorithms such as `DEFLATE`.

By trading off space and accuracy in the form of approximation errors, time-series data compression via PLA can be used as a building block in analysis workflows, allowing larger compression (and thus larger error) for less sensitive analyses and smaller compression in more critical situations. In the LiDAR examples, slightly distorting point clouds in the range of centimeters may be permissible for creating a 3D map of the environment, whereas the precision requirements for collision detection using a point cloud are much stricter.

Continuous Compression

Besides lossy or lossless compression, other aspects are important for the applicability of compression, such as whether the compression scheme requires a single or multiple passes over the data. Several widely used compression techniques fall into the latter category, and need additional passes, for example, to normalize a time series [126] before compressing it. This approach can allow to find optimal representations (with respect to target characteristics such as size) by improving iteratively. For unbounded and fast-paced streaming data in resource-constrained environments like VCPSs, multiple passes may be less suited. To this end, single-pass, continuous compression algorithms that spend a constant time per ingested data point can offer rapid compression at small computational footprints. While single-pass algorithms for compression can not outperform multi-pass alternatives in regards to compressed data size (as multiple passes allow for continued optimization of the compressed representation), single-pass *lossy* compression presents a viable option in VCPSs by trading off accuracy and computational demands [81].

3.3 Provenance in Stream Processing

Complex data processing can involve numerous transformations of raw input data in parallel and in pipelines that are increasingly more complex than simple extract-transform-load (ETL) workflows serving simply to move data from one system to another. As an example from VCPSs, consider the analysis pipeline from Figure 2. Here, data is mapped, aggregated over different windows of times, filtered, joined, and eventually aggregated over many vehicles (and thus many partial pipelines) to generate the final output. In such pipelines, with strong dependencies between various pieces of data in various parts of the pipeline, it becomes more difficult with every additional transformation to understand how and why a certain (intermediate) output was produced. The history of intermediate results and transformations, down to the input data, that eventually leads to the production of an output is also called the latter's *lineage* or *provenance*. Such provenance can exist at different levels of granularity, where *fine-grained* data-level provenance precisely connects pieces of data that are causally related.

There are two main reasons for the interest of analysts in the provenance of a piece of data [78]: (i) first, complex data pipelines, like all complex software applications, usually require extensive debugging and search for errors. The provenance of data can help to discover why a pipeline is not working as expected, or why it is producing erroneous results, or even why expected results are missing [154]. Tracing back the relationships between individual data items to detect where in a pipeline an error originated can be of crucial importance for fixing, improving, or verifying the correctness of an analysis workflow. (ii) Second, provenance can be essential in *explaining* a result in regular operation of the pipeline. In the example pipeline from Figure 2, where final results are the ID and geographic area (A/B) of the fastest vehicle in the last hour, provenance allows analysts to trace back exactly to those initial position and speed readings from an individual car that eventually led to such a result. Knowing that exact input data may enable further analysis to, for example, redefine the geographic areas to balance the speeding reports between areas A and B. Going further than debugging and explaining, when connecting to the idea of the prioritization of relevant data from Section 2.2, *Which data should be prioritized?*, provenance provides a mechanism for selecting relevant input data. This can be achieved by shaping a query such that only relevant input data leads to results, and then delivering this relevant data using provenance.

For short pipelines, or finite input data, provenance can be obtained manually, for example by rerunning a pipeline for different parameters or by altering the input and observing changes in the output. However, keeping track of provenance quickly becomes intractable for complex pipelines and may even be impossible in Stream Processing applications where the input data cannot be stored to be replayed. Even more, such procedures can be too slow in latency-sensitive applications that require provenance as soon as possible to make critical low-latency decisions, or deliver relevant input data quickly. Thus, especially for fast-paced Stream Processing applications, the generation of provenance has to be automated using provenance frameworks.

Fine-Grained Backward Provenance The provenance defined in the preceding examples usually traces from the outputs to the inputs of a query. This directionality is captured in the term *Backward Provenance* [77], [150]. Backward Provenance produces a set of disconnected graphs, where each graph contains on the one hand the result, and on the other all input tuples contributing to that result, making it a useful tool for debugging applications and error-hunting. Backward Provenance for a Stream Processing query can itself be represented as a stream that regularly produces new elements as soon as the underlying query produces an output.

An example is shown in Figure 4 for two queries that process a stream of positional reports of two vehicles (red/mint), with updated reports being produced every five minutes. The first query, Q_a, produces alerts a^i if $^2/_3$ of reports of the last 15 minutes fall into region R, and Q_b produces alerts b^j if the car's mean speed in that time window is greater than 110km/h. The two timelines in the figure show output streams of both queries, carrying the alerts (where the time window leading to the production of the alert is indicated)

and, using fine-grained Backward Provenance, also the input tuples causing these alerts. As shown in the figure, some input tuples (here, t_2^1, t_2^2, t_2^3 from vehicle 2) even contribute to several alerts and are part of multiple graphs.

3.4 Complementary Data Mining Techniques

In this subsection, I will highlight two additional data mining or analysis techniques that are used in the second part of this thesis and that serve as examples of common techniques for mining data from VCPSs and similar systems.

3.4.1 Distance-Based Clustering

Clustering of data is an important problem in data analysis. The core idea is to group data into sets or *clusters*, where the intra-cluster similarity among a certain dimension is maximized. A clustering operation thus assigns every data point in the input set to one of several output sets. Various similarity metrics exist for generating clusters. Here, I will focus on Euclidean distance clustering: the Euclidean distance between two points is the shortest distance between them in the Euclidean space, such as $\mathbb{R}^3$, the space of all real numbers. Various additional parameters can shape the output of a clustering operation, as shown in the following example:

Considering the input data set $\{1, 2, 2.5, 6, 9, 9.5, 10\}$ (ordered for ease of exposition), the clustering using (i) these points' Euclidean distance, (ii) demanding a maximum distance $maxDist$ between clustered points of 3, and (iii) a minimum cluster size of 3, is $\{1, 2, 2.5\}$ and $\{6, 9, 9.5, 10\}$. For $maxDist = 2$, the output would be $\{1, 2, 2.5\}, \{9, 9.5, 10\}$, while $\{6\}$ can be classified as noise (6 has no neighbors at most $maxDist = 2$ away, and as a single point does not fulfill the minimum cluster size).

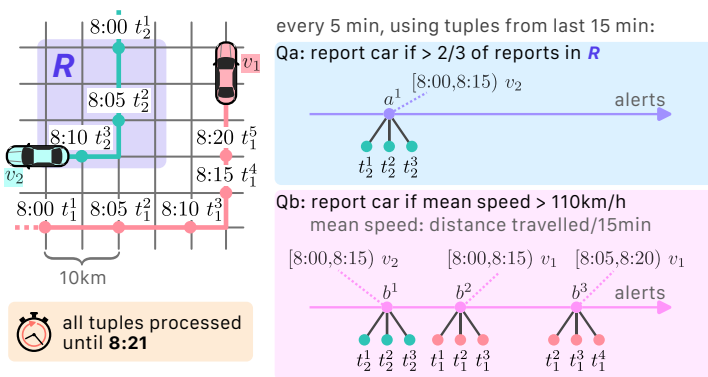


Figure 4: Two queries, Q_a and Q_b, are processing an input stream of positional data tuples from the cars v_1 and v_2. For each query, the stream of fine-grained Backward Provenance is shown as the alerts (outputs) from each query connected to the input tuples that causes the respective alert.

To find an optimal clustering for some parameter choice, it may be required to perform several passes over the input data to perform all required distance comparisons for discovering which data points have minimal distance to which cluster. Such an approach can be slow or infeasible for very large or unbounded datasets. However, if the data structure of the input data admits some ordering, it can be possible to skip certain comparisons. Taking the above example, if the data is delivered in the order $\{1, \mathbf{2}, 2.5, \mathbf{6}, 9, 9.5\}$ and a maximum distance between clustered points of $maxDist = 3$ is given, data point **2** does not need to be compared to any point to the right of point **6**, as any such point will have a distance larger than the $maxDist$ (because the data points are delivered in order). Such order in some dimensions of the input data can even admit a constant number of comparisons per input point, when it is known that there is a minimum distance between two consecutive data points (as for example for a rotating LiDAR, see Chapter A). This finally allows for a processing of data points in a Stream Processing fashion - in these cases, clustering can be performed efficiently even on low-powered hardware on very large or unbounded input datasets.

3.4.2 Federated Machine Learning

In this section, I will give a brief and high-level summary of Federated Machine Learning (FL).

Machine Learning (ML) nowadays commonly refers to *Deep Learning*, in which high-dimensional functions (or ML *models*) with large numbers of parameters are fitted to input data to minimize a loss function. The loss is a measure of the difference between the output of the function, usually a prediction, and the true value for an input. For example, the input could be the image of a bird, the prediction could be "hawk", while the true value is "eagle". In this example, the loss would be greater than zero, since the prediction is not accurate. The fitting process is called *training*, and proceeds by calculating the sum of the loss over a number of samples from the input data for some choice of parameters, before then adapting the parameters such that the loss decreases. This process continues until the loss is satisfactory.

Remarkably, it has been shown (one of the first times in [139]) that the parameters of instances of the same model, trained on independent data sets from the same domain, can be *averaged* over several models to yield a new model. This model then performs good predictions on not only each of the independent data sets, but on the whole domain [139]. This circumstance is subject to several qualifications, such as how similar the independent data sets are and how long the models have been trained before the parameters are averaged. However, in the optimal case, a model for a totality of data generated by averaging over models for subsets of the data is identical to a model for the totality of data generated by training directly on the complete data.

This idea finds concrete use in the sub-field of FL, and can lead to large communication savings. In FL, first, models are trained on subsets of data, and then an averaged model is created and used to replace all models. Then,

these models are trained again on the respective subsets (in this process, their parameters diverge again), a new averaged model is generated, and the replacement occurs again. This process is repeated until the averaged model has a satisfactory loss value, or a certain number of repetitions have been reached. Here, it is unnecessary to have access to the actual input data of the individual models to generate the averaged model, only the parameters of the individual models are required. As one main benefit, the parameters of a model typically have a much smaller size than the data used to generate these parameters - thus, FL can be a technique to reduce the number of data that needs to be exchanged when creating a model.

As an example for a VCPS, a subset of vehicles could train a model[(d)] for some specific use case on data sensed by the vehicle itself, and then the parameters of the model are uploaded to the cloud, averaged over all participating vehicles, and sent back to the vehicles where they replace the *local* models. Thus, possibly significantly less data is exchanged than when the *raw data* is sent by each vehicle to the cloud, where a model is trained over the totality of data, and then said model is sent back to each vehicle. Typically, however, the process of averaging models and sending them back and forth has to be repeated many times, with every such round reducing the advantage of not having to send raw input data. Even in these cases, FL can be advantageous, as it conserves the privacy of every vehicle's driver - only the parameters of a model trained on their data are sent to the cloud, not the data itself. This makes it difficult or impossible to draw conclusions on the actual data used to generate the parameters[(e)].

[(d)] Note that the *training* of ML models is computationally costly, and more computational headroom is required onboard compared to simply using a *pre-trained* model to make predictions. Thus, even modern cars designed to make use of ML models may not be dimensioned to perform onboard ML training.

[(e)] With additional effort, contributions from individual models and thus individual users can be completely obscured [25].

4 Research Methodology

For this thesis, I have followed a set of guiding principles for performing research, where applicable. In this section, I will give a high-level overview of these.

Approach and Formalization

My approach to research began with the identification of data analysis requirements in the E2C continuum with a specific focus on VCPSs, and the subsequent discovery of a gap in existing research. This involved the search for and study of relevant related works to substantiate the identification of a suspected research gap and to identify the required formalisms to reason about the problem appropriately. To cover various parts of the spectrum of possible research problems, I regarded different layers of abstraction:

1. **Complete system perspective**: problems whose scope is a complete system of connected edge nodes and central server.

2. **Sliced system perspective**: problems whose scope is a single edge node connected to the central server.

3. **Pure node perspective**: problems that focus on the processing inside a single node (edge or central server).

It should be noted that these are not mutually exclusive perspectives on problems of distributed data analysis, as, for example, solutions to problems from the pure node perspective (3) extend also to distributed scenarios (1, 2).

The identified problems were then formalized using either existing formalisms, or by introducing novel ones that extend the existing language. In the case of problems involving Stream Processing, my basic formalisms are adapted from the work on the *Dataflow model* [3]. Where possible, I utilized graphical representations of my ideas to help guide both my research and the later presentation.

Analytical and Empirical Evaluation

Solutions were proposed through novel algorithms based on a system design appropriate for the problem. The verification of proposed solutions then occurred first in an abstracted formal manner, followed by an empirical experimental evaluation based on implementations of algorithms.

The formal proofs relied on the usage of theorems and corresponding deductions in a mathematical manner that leverages the earlier formalization of the problem. This step was performed in an iterative fashion, where an adapted problem solution is followed by a formal investigation.

Having formally verified and investigated a proposed solution, I then created an implementation from the solution's algorithm and the suggested system design. This implementation followed the following standards:

- Programming language: Stream Processing algorithms were implemented with Apache Flink [29] due to its large adoption in both research and industry. Performance-relevant system aspects were written in Java, and Python was used where the existing software ecosystem required it as well as for experiment orchestration.
- Goal: the goal of the implementation was to fulfill the algorithmic requirements and be *user-friendly* enough for thorough evaluation (and documented enough for reproducibility, see following subsection), while avoiding the use of additional resources required to bring the implementation to a production level.

The evaluation experiments required input data for the analysis that was to be improved by the solution. When choosing such input data, the following aspects were considered:

- Relevance: Real-world data was used where possible to verify the proposed solutions in real-world applications. Proprietary data from Volvo Cars posed the highest standard for relevance of the solution.
- Availability: To interact with the research community and encourage reproducibility, publicly available data must be included in the evaluations.

Where possible, I thus used both proprietary (to demonstrate the highest levels of relevance) and publicly available datasets (to enable reproduction and further research).

Results Dissemination

For all major results of my research, I strove for publication in peer-reviewed and established conference proceedings and journals, to receive proper feedback from the review process that could be used to improve my work. I included publicly available code repositories for both my implementations and my evaluations where possible. The aim was to design a code repository that is approachable and enables easy reproduction and verification of all experiments and corresponding figures in my publications, requiring additional step-by-step instructions as well as scripts that aide in performing the experiments and figure plotting.

To bring my results from the level of research to industrialization, I disseminated my results, in addition to academic platforms, at appropriate industry fora, and initiated and engaged with work on industrial proof-of-concepts based on my research.

5 State-Of-The-Art and Research Questions

After the introduction into techniques and paradigms that are relevant for this thesis and the presentation of some guiding research methodology, I will now motivate the research questions at the basis of this work. These emerge from requirements posed by E2C continua for fashioning data analysis more efficiently, and from the current state-of-the-art. Noting that distributed, parallel, and low-latency approaches and algorithms are essential in all cases, the research questions cover data compression at the edge, Data Localization and Selection, and the validation of distributed data analysis algorithms in a VCPS.

5.1 Lossy Time Series Compression on the Edge

Moving less data from the edge to the cloud alleviates pressure on the communication infrastructure and can lead to direct monetary savings if usage of carrier-operated cellular networks can be reduced (see Section 2.2). A direct approach for reducing the transferred data volume is to compress the data on the edge before transferring it. As shown in Section 3.2, a host of techniques exist that are specialized for compressing time-series data such as that produced in E2C continua on edge devices. However, given the constraints posed by the latter in the form of small computational capabilities (see Section 2.2), such compression must have a low overhead; and faced with fast-paced continuous data streams, compression that uses only a single pass over the data is advantageous. A compression technique optimized specifically for edge devices is, for example, presented in [81]. Optimizing compression and resource usage, the authors opt for lossy PLA compression (see Section 3.2) with a small memory footprint and instruction count per operation. As lossy PLA produces straight segments that approximate the raw data points, best-fit-line techniques can produce arbitrarily large deviations between the line representation and the raw data. However, in many application such as those that are safety-critical in VCPSs, a certain precision of the data is crucial. Any deviations from the raw data introduced by compression will lead to inaccuracies downstream in the analysis pipeline, and such inaccuracies may render the use of boundlessly lossy compression impossible. In [81], the author's technique bounds the maximum deviation per segment and thus gives some precision guarantees, but an evaluation of the downstream effect of the overall reduced precision was not presented.

It thus remains to investigate how the usage of continuous, lossy compression in a VCPS analysis pipeline with data originating on the edge impacts the eventual accuracy of the analysis. Even more, the trade-offs between size of the compressed data and analysis accuracy must be investigated together with computational overheads to enable analysts to tune compression according to their respective needs.

Thus, I formulate the first research question:

Research Question 1 (RQ 1)

How does continuous lossy time series compression on the edge of a VCPS Stream Processing pipeline affect accuracy and overheads of the pipeline?

5.2 Discovering and Selecting Relevant Data

As underlined in Section 1 and Section 2.2, not all data sources and not all data coming from a specific source may be of equal importance. If the analysis question at hand requires specific types of data, or specific events in the data, it is possible to separate valuable from invaluable data semantically. In this context, I use the terms *Data Localization* and *Stateful In-Stream Data Selection* (or simply, Data Selection) for the process of selecting data or data sources that are of specific interest. Selection of data sources or data itself can ease downstream pressure on the processing pipeline when heavy analysis is performed only on selected data, or it may even be a required part of an analysis pipeline to, for example, anonymize certain pieces of data on the edge before they are transmitted to a central server. Similar to compression (explored in Research Question 1 or RQ 1), it can also reduce the amount of data that must be transmitted and thus ease constraints in network infrastructure and reduce costs for data transmission.

Data Localization

In many systems in the E2C continuum, the sensor set carried by edge devices can be assumed to be known. Thus, the types of data expected from each edge device are known a-priori. With this knowledge, analysis workflows can be designed to include only those edge devices that have the appropriate sensor set, be it when the demands are for the edge devices to transmit raw data, or to perform some analysis already on the edge. However, in many cases, it is possible to designate even finer characteristics of those data sources or edge devices suitable for an analysis than just their general type. As an example, if analysts want to investigate the behavior of vehicles on parking lots in a VCPS, identifying all those vehicles in the VCPS that have been at parking lots can be an important first step of the pipeline. Once these vehicles have been selected, the proper analysis task may then be deployed only to them, or data be requested only from them, instead of broad centralized data collection involving also unsuitable data sources. Such unrefined selection can result in the transmission of data of low value, or excessive data amounts. With data in VCPS being distributed in a highly skewed fashion, a simple random selection of vehicles is not guaranteed to provide a subset of vehicles with relevant data sources [53].

We call the task of selecting edge nodes and relevant data sources *Data Localization*: the identification of edge devices that carry data relevant to an analysis question. The process of Data Localization itself induces overhead on the edge devices, and as such should be designed as efficiently as possible. Special consideration must be taken as Data Localization requests for various

analyses could be deployed simultaneously over the same edge devices. Simultaneously, it is desirable that Data Localization can be performed quickly, so that any additional latency introduced is minimized.

There is a body of work for general querying for data inside vehicular networks, usually focusing on vehicle-to-vehicle communication inspired by peer-to-peer approaches [129], [197] or relying on communication with road-side units [5], [51]. However, the specific usage of the architecture of the network in these works, relying, for example, on advantageous positioning of road-side units, can fail to translate to modern VCPSs that lack V2V and V2X connectivity. In [53], the authors introduce a continuous querying mechanism that localizes relevant data in a fleet, but it does not aim at reducing the overall number of queried vehicles and thus may introduce cumulative overheads on the fleet that can be further minimized.

With this background, I pose the following second research question:

Research Question 2 (RQ 2)

How can we in a VCPS select relevant data sources while balancing cumulative computational overhead and duration of the localization procedure?

Stateful In-Stream Data Selection

In systems with constantly evolving data streams, Stream Processing offers the possibility to filter data via the Filter operator directly (introduced in Section 3.1). This allows some basic selection of relevant data. However, the Filter operator is *stateless*, operating only on a per-data-point basis, and does not take previous data points into account for selection. Thus, the Filter operator must be used in combination with other, possibly stateful, operators, to create the precise selection mechanism required for the analysis at hand. In that case, however, all data not passing the Filter will be discarded, while data that does pass the Filter is aggregated and may thus not be in the shape required for further analysis.

Backward Provenance in Stream Processing, here introduced in Section 3.3 and realized efficiently in [151], can deliver all input data to a Stream Processing pipeline that causally leads to an output from the pipeline[(f)]. As explained in Section 3.3, a Stream Processing pipeline can be tailored such that only relevant input data generates an output. Then, using Backward Provenance, one can effectively select the relevant input data, even for complex, stateful, and concurrent queries.

For fine-grained Data Selection, however, Backward Provenance presents several drawbacks. First, it potentially delivers source data multiple times (if a piece of source data contributes to multiple outputs; see, e.g., input tuple t_2^1 in Figure 4 that contributes to results from two queries), and second, it omits *temporal* information from the provenance data. As an example, if a

[(f)] Generally, this holds not only for input data to the pipeline itself but also for intermediate streams of data inside the pipeline. For ease of exposition, we discuss here only the first case, but note that the ideas hold for intermediate data as well.

piece of input data timestamped 10:00 contributes to a result at 10:05 and is then *selected*, such selection could be premature if the same piece of input data contributes to another result at 10:10 due to the input data's lifetime in the pipeline. Such multiple contributions can, for example, enable a more complex prioritization among selected data. An example is found in Chapter C, with a Stream Processing query in which the input data consists of camera frames from a vehicle. In that example, there are two types of query outputs, namely alerts on crossing pedestrians or cyclists in front of the vehicle. If now an input frame contributes to a pedestrian alert first, and then later to a cyclist alert (e.g., because the cyclist pipeline runs slower), any decision taken before the second contribution may be premature. Backward Provenance does not indicate when it is safe to make a decision and thus omits the tuple's *liveness*.

In a nutshell, the need for a more potent In-Stream Data Selection mechanism in Stream Processing is summarized in the following third research question:

Research Question 3 (RQ 3)

How can we efficiently and in a stateful fashion select relevant data within a set of Stream Processing sources in a live and duplicate-free manner?

Cormode, Garofalakis, Haas, *et al.* have stated that *"[I]t has been increasingly realized that extracting knowledge from data is usually an* interactive *process, with a user issuing a query, seeing the result, and using the result to formulate the next query, in an iterative fashion"*. Especially against the background of ever-evolving patterns in data streams and changing analysis questions, constant adaptation of deployed queries is necessary to gain the desired insight and to adapt to changes in the data. From these statements it follows that Stateful In-Stream Data Selection should be provided not only live and without duplicates, but even for *evolving* sets of queries. In such evolving sets, queries can be added and removed at run time of the system without interrupting the execution. State-of-the-art Backward Provenance [151], while enabling some features of Data Selection as outlined above, works only for *static* sets of queries that are known a-priori and is thus unfit also in the additional criterion of dynamicity to provide Data Selection for modern analysis workflows.

It can be expected that additional complications arise from providing indications of the selected data's liveness when input data can contribute to results of queries added during the run time of the system, as such a system has to constantly adapt to the changing query conditions. At the same time, queries that analysts want to add have to be admitted as quickly as possible (and, likewise, if an analyst wants to remove a query), to keep additional latencies minimal and the system responsive.

From this, I formulate my next research question:

Research Question 4 (RQ 4)

Can we enable Stateful In-Stream Data Selection within dynamic sets of Stream Processing sources for modern analysis workflows?

5.3 Evaluating Novel Vehicular Distributed Data Processing Algorithms

The research questions posed in this work touch both upon more isolated and sliced views of single or few edge devices connected to central servers as well as large networks with many edge devices. In the former case, challenges of evaluating and validating research approaches and frameworks may involve the simulation of network conditions (Chapter A) and the usage of appropriate hardware stand-ins for VCPS devices (Chapters C, D) to various extents. Challenges posed from these experimental settings, however, are compounded when working with the assumption of many edge devices communicating in various ways with the central server and possibly each other. Even more, in the perspective of a larger system such as a VCPS, devices may lose connectivity because of their mobility, may be turned off and thus leave, or turned on and join the edge. Such phenomena of intermittent communication and device turnover or *churn* can critically impact the behavior of any algorithm designed to disseminate information or distribute communication in VCPSs. To help to explore RQ 2 in a more realistic setting, for example, we must assume that the members of the network change during the execution of a data localization algorithm.

There exist separate frameworks for simulating wireless communication [31], [189] and also frameworks that simulate movement of agents in traffic [67], [131], as well as approaches that combine communication and movement [171], [178]. With access to real-world trajectory data from vehicles, it is, however, possible to validate VCPS settings using even more realistic assumptions by replaying actual vehicle data, furthermore, the artificial generation of movement data can present unnecessary overhead. In addition, using layers of abstraction for communication instead of a full fine-grained communication simulation can facilitate the experimentation setup and make it more approachable, enabling experimentalists and researchers to focus on the distributed algorithm to be tested.

With these qualifications, I raise my final research question:

Research Question 5 (RQ 5)

How can we validate distributed data processing algorithms for the realm of VCPSs efficiently?

6 Thesis Contributions

Having introduced research questions that guide this work, this section now outlines the contributions contained in each of the main chapters in addressing those questions. For a quick overview, the following table provides a mapping between the research questions and the chapters:

Mapping research questions to chapters

	Chapter A Edge Compression	***Chapter B*** Data Localization	***Chapter C*** In-Stream Data Selection	***Chapter D*** Dynamic Data Selection	***Chapter E*** Distributed Analysis Evaluation
RQ 1	✓				
RQ 2		✓			
RQ 3			✓	✓	
RQ 4				✓	
RQ 5					✓

6.1 Low-Overhead Tuneable Lossy Time Series Compression on the Edge

In response to RQ 1, *"How does continuous lossy time series compression on the edge of a VCPS Stream Processing pipeline affect accuracy and overheads of the pipeline?"*, I have in Chapter A designed a complete Stream Processing pipeline for data originating on a vehicle. This data is sent wirelessly to a central server. There, it is *clustered* (see Section 3.4.1) in an online fashion, enabling use cases such as the generation of 3D environments from LiDAR data or city maps that track traffic density over time. Online lossy compression is introduced as a module running on the vehicle that continuously generates a smaller representation of the data streamed to it before the compressed data is transmitted. At the central server, the data is received and decompressed in an online fashion before it is continuously clustered.

The contribution of this work consists of three parts. First is the adapted online clustering algorithm, which is based on [142], a clustering algorithm for LiDAR point cloud data that has a constant overhead per data point and thus enables clustering in an online fashion. In the chapter, this algorithm is extended to process any type of data in which the data structure exposes the dimensions that are relevant for clustering[(g)].

[(g)] An example: for rotating LiDARs, points that are generated close in time can have a bounded difference in space, which is then mirrored in the timestamp-sorted data structure or data stream. When clustering LiDAR data with distance-based clustering, thus the data structure already partially exposes the distance between points.

The second and main contribution is the development of a lossy piece-wise linear (PLA) online compression algorithm (here implemented in the SPE Apache Flink [29]) that gives a maximum error bound for each compressed data point, has an amortized constant overhead per data point, and allows the independent compression of time and signal channels for time-series data. Using as input parameters only the maximum length of each compressed segment and the maximum error bound allowed, a tuneable trade-off between accuracy and size of the compressed representation is offered. This algorithm is shown in the chapter's evaluation section to run with low overheads even on hardware representative of the computational headroom on modern vehicles.

The final contribution is the extensive evaluation of the effect of maximum error bound on (i) the faithfulness of compressed to raw data, (ii) the data size, (iii) the duration of the data transmission for simulated network speeds (where stronger compression results in smaller data sizes that are faster to transmit, but take longer to generate because of higher computational overheads), and (iv) the accuracy of the final clustering analysis. The evaluation shows that for the presented use cases and raw data types, the maximum error bound can be tuned such that data size reductions are substantial while accuracy losses remain small. In the use cases presented, the algorithm achieves data sizes that are only between 5 and 35% of the raw data size, while the clustering analysis retains over 90% of its accuracy. For almost-lossless compression (in which the faithfulness of the compressed data is high), the presented PLA compression scheme outperforms common DEFLATE compression (used by `zip`) by factors of 2 to 10 in data size.

The compression algorithm in this work has been tested in real automotive environments at Volvo Cars and is, as of this writing, used in production on-board of test vehicles to compress data before transmitting it, furthermore validating its applicability in real-life scenarios.

To summarize, the work in Chapter A answers RQ 1 by presenting a novel online lossy compression algorithm that allows to balance between the compression size and the analysis accuracy. The downstream effect of this algorithm is extensively evaluated along a range of use cases, illustrating one path to ease the latency and network bottlenecks presented by data transmission from the edge to the cloud.

6.2 Data Localization

RQ 2 asks *"How can we in a VCPS select relevant data sources while balancing cumulative computational overhead and duration of the localization procedure?"*. To answer this question, I have in Chapter B designed a family of algorithms to efficiently distribute *requests* for Data Localization in VCPSs. I assume such VCPSs to consist of vehicles that each store a different and unknown set of data of varying size and composition, and a central server at which requests for a minimum number of data sources fulfilling some *condition* are issued. The condition is sent out from the central server to a subset of vehicles which check whether the condition holds on their data and reply with an answer consisting only of *yes* or *no*. Every vehicle answering positively is counted as one localized

data source. In this scenario, the role of the algorithm family I introduce is to find N vehicles on which the condition holds. However, every vehicle contacted by the central server with a request will spend some unknown amount of work searching its data (this amount depends on the condition to be checked and the local data amount).

In this work, I attempt to minimize the cumulative work done by the fleet resulting from checking query conditions and do not focus on search optimizations on-board the vehicles. To achieve this minimization, the algorithms should avoid selecting an excessively large subset of vehicles that are tasked with checking the condition. Simultaneously, the algorithms should find N suitable vehicles as quickly as possible. I present four algorithms overall that position themselves at various points along the axes of time and workload, with two of them posing as baselines. The first is an *eager* algorithm that sends the request to every available vehicle, resulting in maximum load but minimum overall duration. The second, a *lazy* algorithm, is proceeding in rounds, asking a minimum number of vehicles each round and proceeding once the asked vehicles have responded, with potential further rounds to discover remaining data sources. This latter algorithm is likely to ask a minimum number of vehicles, but may take maximum time in the worst case.

The main contribution described in this chapter lies the remaining two algorithms, the *balanced* and *fair* algorithm. These position themselves between the two baselines and can be parameterized to yield desirable results for both speed *and* total workload. These algorithms both proceed in rounds, using knowledge gained in earlier rounds about the share of vehicles that hold relevant data as wells as mechanisms to start new rounds prematurely before all contacted vehicles have responded. In introducing the algorithms, I present different models of the fleet, ranging from a synchronous, static model in which communication is instantaneous and all vehicles require equal time for checking the query condition, to an asynchronous, dynamic model in which the vehicles dynamically join and leave the fleet, query checking times vary depending on data and composition on each vehicle, and communication is modelled with a distribution of varying latencies. To model realistic demands for Data Localization, I evaluate up to 15 simultaneous Data Localization requests (all requiring different conditions to be checked) that are deployed concurrently over the vehicle fleet, with vehicles involved in multiple requests answering those in a first-in-first-out fashion.

The evaluation, performed on hardware representing available computing capacity on modern vehicles, using two real-world data sets, shows how the *balanced* and *fair* algorithms can achieve speedups of up to 40 times compared to the *lazy* baseline, while consuming only a third of the resources of the *eager* baseline. A Data Localization procedure using the algorithms proposed in Chapter B may thus be used as a first step in an analysis pipeline that removes the necessity of involving excessive amounts of vehicles in an analysis. Thereby, the number of irrelevant data transmitted (when the next step in the pipeline is data transmission) or of work done by the vehicles unnecessarily (when analysis is to be pushed to the edge) can be significantly reduced.

In the process of localizing vehicles with suitable data, the algorithms

dynamically update their knowledge of the data distribution. As I additionally show in the evaluation of Chapter B, this results in approximations of the data distribution that are obtained as a by-product of the localization procedure but can stand by themselves as important insights about data in the fleet.

In summary, Chapter B answers RQ 2 with a family of algorithms that provide a tuneable knob to analysts for localizing data in a VCPSs while balancing the cumulative computational overhead and the overall duration of the procedure.

6.3 Stateful In-Stream Data Selection

To answer RQ 3, *"How can we efficiently and in a stateful fashion select relevant data within a set of Stream Processing sources in a live and duplicate-free manner?"*, I take in Chapter C the idea of Backward Provenance in Stream Processing (see Section 3.3) as a starting point to develop the notion of *Forward Provenance*: Forward Provenance provides a bipartite contribution graph connecting in a Stream Processing query all inputs with those outputs they have contributed to. This graph can directly be traversed forwards (from inputs to outputs), but just as well backwards. It is free from duplicates (unlike in Backward Provenance, which delivers a sequence of disjointed graphs that may contain inputs multiple times, should they have contributed to multiple outputs) and furthermore *live*, containing special labels on each input and output that indicate which parts of the graph are still subject to change. As an example, if an input contributes to a first output, but effects from said input reverb in the Stream Processing query *after* the first output is produced (because the input contributes to intermediate results that are still in processing), it is possible that further contributions in the form of more output tuples are created in the future. This contribution window closes once all intermediate results that the input has contributed to are not in processing anymore. Forward Provenance recognizes this and provides an upper bound on when the contributions from an input are actually finalized, taking into account the lifetime of the contributions of an input across an arbitrary number of parallel queries that all process the same input stream. Once contributions from an input are finalized, the neighborhood of said input in the graph will not change anymore in the future, providing an important guarantee for further analysis of the contribution graph.

As such, Forward Provenance is suited to not only advance well-discussed applications of Backward Provenance such as debugging, but provides a tool for *selecting* data in Stream Processing pipelines. Tailor-made queries can ensure that relevant data becomes part of the contribution graph via Forward Provenance, the absence of duplicates leads to an overall data reduction, and the live properties of the graph ensure that further processing on selected data occurs only once the provenance information about that data is final. As an advanced example application, the results of a Stream Processing query can serve as annotations for the inputs (that are obtained by traversing the graph from the input to the results), and by having multiple types of results that the input can contribute to (from parallel pipelines), Forward Provenance can

deliver multiple annotations per input tuple that can become processed once the input is finalized to evaluate its overall importance.

In Chapter C, after defining and reasoning about Forward Provenance in Stream Processing, I present a framework called Ananke that is based in Apache Flink and that provides all aforementioned properties of Forward Provenance. I prove its correctness, present a custom implementation and one based solely on out-of-the-box Stream Processing operators, and fully evaluate the Ananke framework along a range of real-world and benchmark use cases. The evaluations show that additional overheads from Forward Provenance for the most efficient implementation range from 0 to 14%, and I furthermore evaluate Ananke's performance against database solutions that create the Forward Provenance graph on-demand, incurring significantly higher latencies.

From Static to Dynamically Evolving Sets of Queries I introduce Forward Provenance in Chapter C for sets of Stream Processing queries that are static over the whole run time, meaning that the composition of queries for which Forward Provenance is provided does not change. Evolving from this, Chapter D introduces the notion of Forward Provenance for *dynamic* sets of queries, where a) there exists no a-priori knowledge of the queries when the system is started, and b) queries are added and removed from the set of queries for which Forward Provenance is provided upon user requests. As such, this work addresses RQ 4, *"Can we enable Stateful In-Stream Data Selection within dynamic sets of Stream Processing sources for modern analysis workflows?"*.

I answer this question positively in Chapter D and formalize the requirements for Forward Provenance under *dynamic query sets*. I first demand that the guarantees provided under static conditions hold in the dynamic scenario. Then, I describe how queries can be added to and removed from the set of queries in question at run time while continuously providing liveness information in the graph. Crucially, I present how this can be done without re-activating input data that was previously marked as final by re-feeding such input data to newly added queries. Eventually, I present Nona, the first framework that provides Forward Provenance in Stream Processing in such dynamic scenarios, and prove that it delivers on the presented requirements.

I provide an extensive evaluation of Nona, using real-world and benchmark data on hardware on both ends of the E2C continuum, from single devices over low-power clusters to a powerful manycore server. The evaluation exemplifies the architectural overheads necessary for adding and removing queries at run time to and from a Stream Processing system as well as for providing Forward Provenance in these cases. While the architectural requirements partially lead to better performance at larger resource utilization, the added overheads for providing *dynamic* Forward Provenance are shown to be minimal. Furthermore, I show that Nona accommodates the addition and removal of queries within only a few hundreds of milliseconds, all while continuously providing Forward Provenance for every deployed query.

With this contribution, I enable the selection of data in Stream Processing queries for modern workloads, in which queries are added, removed, and re-deployed in response to changes in requirements and novel discoveries from the

data.

6.4 Validation of Distributed Data Processing Algorithms for VCPSs

RQ 5 asks, *"How can we validate distributed data processing algorithms for the realm of VCPSs efficiently?"*. I approach this question in Chapter E, where I contribute a complete requirements list for any tool that can stand as an answer to RQ 5. I base this requirements list on concrete experiences gathered at Volvo Cars, related work, and my own observations from performing research involving such algorithms (shown e.g. in Chapter A and Chapter B). The resulting prioritization of characteristics around the modelling of the vehicle fleet, support for modern data processing frameworks (especially from the field of Machine Learning), communication simulation, metrics provision and flexibility demands tools that have hitherto not been presented in the literature. Outlining a proposal for the architecture of such a tool, I finally present Roadrunner, a prototype implementation fulfilling the requirements for validating distributed data processing algorithms in VCPSs efficiently.

Furthermore, to evaluate the possibilities presented by *Roadrunner*, I contribute a novel distributed Machine Learning algorithm that extends the assumption of star-shaped topology usual for Federated Learning (see Section 3.4.2) with the notion of vehicle-to-vehicle communication. As shown in the chapter, such an algorithm leads to a more accurate Machine Learning model with the same communication budget that traditional Federated Learning requires. The evaluation, leveraging real-world data, showcases Roadrunner's capabilities by providing fine-grained metrics at every point in the simulation, where Roadrunner replays GPS traces, coordinates the training of Machine Learning models on powerful hardware similar to that required on modern vehicles, simulates the communication between vehicles and a central server and collects and aggregates metrics.

With tools such as Roadrunner, future focus can be placed on the developments of such distributed algorithms rather than on the tools required to evaluate them.

7 Conclusions

This thesis proposes and evaluates a host of ideas and techniques for tackling the challenges of analysing data in VCPSs with their inherent challenges of imbalanced data generation and distributed computation capabilities. As the growth in data continues to outpace the growth in mobile bandwidth and storage, and cost and power concerns cement the computational capability gradient from edge to cloud, distributed algorithms and Stream Processing are explored in this work to help in resolving analysis bottlenecks. The resulting frameworks and algorithms are in many cases agnostic of the specific analysis at hand, focusing instead on the transmission and selection of data *before* the final data mining step occurs, or the matter of how novel distributed algorithms can be tested in VCPSs.

In the following, I will present conclusions from and future directions for the four main chapters:

Low-Overhead Tuneable Lossy Time Series Compression on the Edge

Chapter A shows that for many pipelines from edge to cloud, a balance can be found that significantly reduces the data volumes that need to be transmitted using novel and low-overhead lossy compression while retaining a high analysis accuracy. This enables analysts to either gather more data from more vehicles, leading to improved insights, or to spend less on data transfer and benefit from the greater speed of the analysis pipeline.

As the PLA representation of raw data can be constructed with small additional overheads, one avenue for future work is the application of PLA compression at an even earlier stage inside a VCPS: in fact, each vehicle itself can be regarded as its own decentralized system, with hundreds of networked sensors around central processing units. Pushing compression even further to the edge and thus close to those small sensors could allow for better usage of the bandwidth *inside* the vehicle, while still allowing to keep the reduced data footprint for wireless communication with a data center. This would require experimentation with even lower-powered hardware than in Chapter A to investigate whether additional overheads could be sustained by such low-powered devices. Insights from this work would extend to similar systems such as modern factories with networks of sensors that continuously monitor the production process.

Data Localization

The Data Localization algorithms in Chapter B balance between computational overheads induced on the edge devices and the time needed for the localization procedure, enabling analysts to discover data both quickly and efficiently, as the evaluation shows. With such Data Localization as the initial step of any data gathering or analysis pipeline, the amount of excessive or irrelevant data and work inside a VCPS can be significantly reduced, freeing up resources for more workloads deployed in parallel over the fleet, and saving communication

costs.

While the proposed algorithms are adaptive, gathering knowledge of the distribution of data on the edge devices per Data Localization request, they do not yet take into account correlations between requests. Estimating the strengths of such correlations can be one avenue for future exploration, essentially re-using knowledge gathered in past requests to optimize the execution of future ones. Our results are based on static data on the edge devices, but the inclusion of dynamically evolving data on the edge could make such re-use of knowledge more challenging, as knowledge of the past may become less relevant as the data evolves.

Stateful In-Stream Data Selection

Chapters C and D introduce and then extend the notion of Forward Provenance in Stream Processing to enable Stateful In-Stream Data Selection even for dynamically evolving sets of Stream Processing queries. The presented frameworks allow analysts to focus on writing stateful queries that create alerts on relevant data, and then to obtain this relevant data at low overheads in a format that makes processing it safe (through guarantees on liveness) and efficient (through the structure of the provided graph).

To provide deeper functionalities of Data Selection, one avenue is to regard the results of a Stream Processing query, which Forward Provenance connects with the contributing inputs, as labels on said inputs. Multiple concurrent queries, producing various types of results, can thus be seen as providing a variety of labels to the input data, that can be combined once input data is final to provide a single- or multidimensional scoring for the data. These input data annotations can be used to prioritize data among several dimensions, allowing for example to keep the most recent and most relevant data in a buffer while continuously discarding old and low-priority data. With the capability of removing and adding queries, the labels given to particular types of inputs can dynamically change over time to adapt to changing requirements.

Evaluation of Distributed Analysis Algorithms in VCPSs

The novel VCPS simulation and evaluation framework presented in Chapter E allows researchers to focus on the distributed analysis algorithm at hand. Thereby, the development of, for example, state-of-the-art distributed ML algorithms leveraging the structure of the VCPS network is enabled. This, in turn, can help to further reduce the amount of data communicated in VCPSs when training ML models, and thus open up more use cases.

Simulating compute-intensive processes such as distributed Machine Learning, in which many vehicles train Machine Learning models possibly simultaneously, could be sped up significantly by extending the simulation tool to leverage not only single but multiple GPUs, with each one used for executing the current work of a single vehicle. Especially for larger and more compute-intensive models, such an extension will be rewarding.

Part II

Main Chapters

Chapter A

Investigating Tradeoffs from Lossy Time Series Compression at the Edge

Bastian Havers, Romaric Duvignau, Hannaneh Najdataei, Vincenzo Gulisano, Marina Papatriantafilou, Ashok Chaitanya Koppisetty

The following is an adapted version of the work published in *Future Generation Computer Systems, Vol. 107, p. 1-17*, as *"DRIVEN: A framework for efficient Data Retrieval and clustering in Vehicular Networks"*. Any changes serve only to retain the consistency of this thesis.

Abstract

The growing interest in data analysis applications for Cyber-Physical Systems stems from the large amounts of data such large distributed systems sense in a continuous fashion. A key research question in this context is how to jointly address the efficiency and effectiveness challenges of such data analysis applications.

DRIVEN proposes a way to jointly address these challenges for a data gathering and distance-based clustering tool in the context of vehicular networks. To cope with the limited communication bandwidth (compared to the sensed data volume) of vehicular networks and data transmission's monetary costs, DRIVEN avoids gathering raw data from vehicles, but rather relies on a streaming-based and error-bounded approximation, through Piecewise Linear Approximation (PLA), to compress the volumes of gathered data. Moreover, a streaming-based approach is also used to cluster the collected data (once the latter is reconstructed from its PLA-approximated form). DRIVEN's clustering algorithm leverages the inherent ordering of the spatial and temporal data being collected to perform clustering in an online fashion, while data is being retrieved. As we show, based on our prototype implementation using Apache Flink and thorough evaluation with real-world data such as GPS, LiDAR and other vehicular signals, the accuracy loss for the clustering performed on the gathered approximated data can be small (below 10%), even when the raw data is compressed to 5-35% of its original size, and the transferring of historical data itself can be completed in up to one-tenth of the duration observed when gathering raw data.

A1 Introduction

Large distributed Cyber-Physical Systems (CPSs) such as vehicular networks [205] (among others) are behind many of the current research threads in computer science. One of the aspects many of such research threads share has its roots in the large amounts of data sensed continuously in large distributed CPSs. As discussed in the literature, the benefits and possibilities CPSs' data enables (e.g., online congestion monitoring, platooning and autonomous driving in the case of vehicular networks) are bound to many challenges, spanning efficient analysis [150], efficient communication [112], [216], security [170] and privacy [86]. A key aspect in this context is the need for solutions that can jointly address several such challenges [82], since solutions that focus on and/or excel in only one aspect but fall short in others might be impractical in real-world setups.

A1.1 Challenges

When focusing on aspects such as data communication and analysis, a well known challenge is given by the imbalance between the amounts of data sensed and produced by the sensors deployed in such CPSs (a modern vehicle, on the road today, senses more than 20GB/h of data [44]) and the infrastructures' capacity of gathering them within small time periods to data centers [59]. Even when data is not to be transmitted continuously, but only for a limited time period and for some selection of sensors, the required bandwidth may far exceed the available one (e.g., a single LiDAR sensor of an autonomous car produces around 7MB/s, cf. Section A4.1). In this case, solutions focusing on efficient data analysis need to account for communication aspects too, in order for the latter not to result in a major bottleneck. The inherent limitations of traditional batch and store-then-process (DB) analysis techniques, which on their own cannot sustain the data rates of relevant applications, need thus to be overcome by taking into account the end-to-end transformation process of raw data into valuable insights. Specifically, considering which data – as well as how much data – is moved through a certain analysis pipeline. Because of this, a complementary challenge gravitates around how to take advantage of the high cumulative computational power of CPSs' edge sensors and devices, since the porting of a given sequential analysis tool (e.g., clustering) to an efficient parallel and distributed implementation and its deployment are not trivial.

A1.2 Contributions

We present the DRIVEN framework, which copes with the aforementioned challenges for a common problem in vehicular networks' applications, namely that of gathering and clustering of vehicular data. In a nutshell, the DRIVEN framework jointly addresses the challenges of data gathering, online analysis and leveraging of edge devices' computational power by:

1. leveraging a lossy compression technique, based on Piecewise Linear Approximation (PLA), that significantly reduces the amounts of data to be gathered from vehicles,

2. leveraging state-of-the-art online clustering techniques such as Lisco [142], which overcome the limitations of batch-based ones, and
3. relying on the data streaming paradigm to transparently achieve distributed and parallel deployments.

As we further elaborate in the remainder, a data analyst interested in gathering and clustering data sensed by a set of vehicles over a given period of time can do so by specifying parameters about (i) the type of data to be gathered, (ii) the maximum error that can be introduced while compressing the data to be retrieved (because of the PLA-based compression) and (iii) the specifications for the clustering of data. The DRIVEN framework then compiles this information into a streaming application that is deployed both at the vehicles providing the data as well as at the analyst's data center. To support modularity, the framework also allows the analyst to define additional components for the resulting application that can be used to process the data before the latter is clustered.

An extensive literature exists about clustering, its porting to the streaming protocol and the leveraging of approximation techniques to improve (along with certain criteria) the clustering process, as we discuss in Section A6. In this context, our contribution does not aim at surveying all existing solutions nor at comparing them. Rather, the contribution focuses on providing evidence of how a streaming application that can (i) jointly leverage the computational power of both edge and central components of a CPS and (ii) allow for partial data loss when gathering information can provide a healthy tradeoff between data reduction and pipeline speed on the one hand and accuracy loss on the other, despite requiring more data processing components (e.g., to compress and decompress the data gathered from the vehicles) than a centralized counterpart (which needs all the raw data to be gathered). As we show in our empirical evaluation, based on a prototype implementation using Apache Flink and recently proposed streaming-based PLA and clustering methods, and four real-world use cases, DRIVEN is able to reduce the duration of data transmission by up to 90% while incurring a bounded loss on the clustering quality. The rest of the paper is organized as follows. We introduce preliminary concepts in Section A2 and the considered system model and problem statement in Section A3. We then present the DRIVEN framework in Section A4 and our evaluation in Section A5. Finally, we discuss related work in Section A6 and conclude the paper in Section A7.

A2 Preliminaries

We begin this section by discussing preliminary concepts about data streaming, PLA, distance-based clustering and logical latency.

A2.1 Data Streaming

The data stream processing paradigm (aka data streaming) [180] emerged as an alternative to the traditional *store-then-process* one. Thanks to its fast evolution

over the last decades, modern Stream Processing Engines (SPEs) allow for distributed, parallel and elastic online analysis [29]. At the same time, efficient designs and methods are in focus in the literature for computationally-expensive streaming analysis [84]. As discussed in [180], the data streaming paradigm has been defined to take into account the challenges proper of large systems gathering data through millions of sensors (as discussed in Section A1). Thus, many applications rely on it in many CPSs, including vehicular networks [12], [46], [147].

In data streaming, each sensor produces a *stream* of data, a sequence of *tuples* that share the same *schema* composed by *attributes* $\langle y^0, y^1, \ldots, y^k \rangle$, where y^0 is a physical or logical timestamp and the other k attributes depend on the sensor producing the stream. We assume that each stream delivers tuples in order based on y^0 as in [18], [108] (or leverages sorting techniques such as [104], [207]). Streaming applications, also referred to as *continuous queries* (or simply queries, in the remainder) are defined as Directed Acyclic Graphs (DAGs) of streams and operators. Each operator defines a function that manipulates its input tuples and potentially produces new output tuples, while streams specify how tuples flow among operators. Modern SPEs such as Apache Flink [29], which we use to implement the DRIVEN framework, provide many operators that can be composed into queries (and also allow for users to define ad-hoc operators). It should be noted that streaming operators are expected to enforce one-pass analysis [180] and can temporarily maintain a *window* of the most recent tuples when an aggregation function (such as clustering) is to be performed on them [84]. As mentioned in Section A1, space and time complexity reduction through approximation and/or partial data loss have been discussed in many flavors in the context of streaming applications. Proposed solutions include load shedding, sketches, histograms and wavelets [16], [17], [49], [182]. In DRIVEN, we rely on PLA, further discussed in the following section.

A2.2 Piecewise Linear Approximation

Computing a PLA of a time series is a classical problem that aims at representing a series of timestamped points by a sequence of line segments while keeping the error of the approximation within some acceptable error bound. We consider here the *online* version of the problem, with a prescribed maximum error Δ, i.e., (i) the time series is processed one point at a time, the output line segments are produced along the way, and (ii) the projected points along the compression line segments always fall within Δ from the original points. Figure A1 gives an example of a PLA: original data points (crosses on the figure) from the input stream are compressed and forwarded on the output stream as line segments (solid lines) or singletons (squares), so that a reconstructed stream (bullets and squares) can be generated from the PLA of the original stream (we refer the reader to Section A4 for more details about why both segments and singletons are defined and the conditions upon which they are forwarded).

In the extensive literature dealing with such an approximation (among others [22], [64], [111]), it is clearly stated that the approximation's main intent

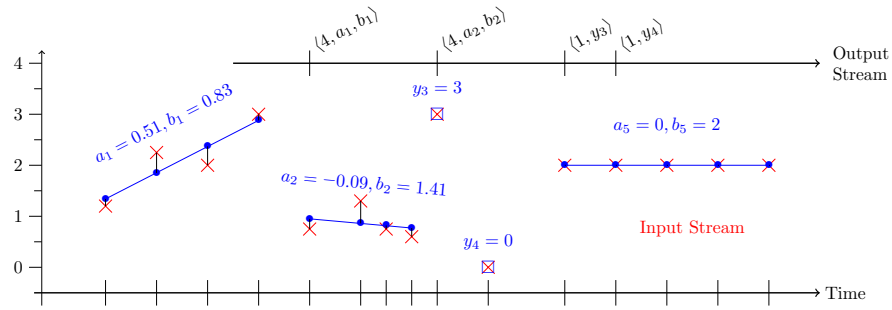


Figure A1: Example of a Piecewise Linear Approximation using maximum error $\Delta = 0.5$.

is to reduce the size of the input time series for both efficiency of storage and (later) processing. This entails a practical trade-off between a bounded precision loss and space saving for the time series representation. Recent works on PLA [58], [81], [134], [201] increasingly place the focus on the streaming aspect of the compression process, and advocate low time/memory consumption as well as small latency while achieving a high compression, in order for PLA to be feasibly implemented on top of, or close to, a sensor's stream.

In this work, we use a best-fit line approximation together with a streaming output mechanism, both introduced in [58] and briefly described in Section A4.2, balancing trade-offs associated with PLA in a streaming context.

A2.3 Distance-Based Clustering

Clustering is a core problem in data mining; it requires to group data into sets, known as clusters, so that intra-cluster similarity is maximized. There are various clustering methods that use different similarity metrics. Among them, *distance-based clustering* methods are able to discover clusters with arbitrary shapes and form the clusters without a-priori knowledge about their number [87]. For ease of reference we paraphrase the definition of distance-based clustering from [172]:

Definition A1. [Distance-based clustering] *Given n data points, we seek to identify an unknown number of disjoint clusters using a distance metric, so that any two points p_i and p_j are clustered together if they are* neighbors*, i.e., if their distance is within a certain* threshold*. To announce the set of points as a cluster (rather than noise), its cardinality should be at least a predefined number of points minPts.*

In a recent work [142], distance-based clustering (for the Euclidean distance case) is studied in the data streaming paradigm to introduce a new approach, named *Lisco*. This approach enables the exploitation of the inner ordering of the data to maximize the analysis pipeline in order to facilitate the extraction of clusters and contribute to real-time processing. In this paper, we use and adapt Lisco as the clustering approach to shape clusters based on distance

similarities without knowing the number of clusters in advance. We discuss more details of Lisco and its adaptations in Section A4.3.

A2.4 Logical Latency

In data streaming literature [84], the term latency usually refers to the (physical) time difference between the production of an output tuple and the processing of the last input tuple contributing to (or triggering the creation of) the former.

This latency definition is usually employed to evaluate the processing performance of a given streaming-based solution. When sequences of multiple input tuples are aggregated together following the processing of a later tuple without an a-priori known size for the length of such sequences (as in a PLA segment, given that the length of each segment depends on the points it approximates), users can also be interested in the *logical latency* introduced by the aggregation mechanism. We refer to the notion of *logical latency* as the number of tuples processed between a given tuple and the first tuple that triggers the aggregation of the sequence to which the former belongs. More concretely, with a sequence of n tuples being aggregated together $\langle y_\ell^0, y_\ell^1, \ldots, y_\ell^k \rangle$, ..., $\langle y_{\ell+n}^0, y_{\ell+n}^1, \ldots, y_{\ell+n}^k \rangle$ and $\langle y_j^0, y_j^1, \ldots, y_j^k \rangle$ the tuple that triggers their aggregation (with $j \geq \ell + n$), the logical latency for any tuple $\langle y_i^0, y_i^1, \ldots, y_i^k \rangle$ is $j - i$ for $i \in [\ell, \ell + n]$.

A3 System Model and Problem Statement

We consider systems consisting of a set of many vehicles and one *analysis center*, in which data analysts are interested in gathering data from such a set of vehicles and, subsequently, clustering that data at the analysis center. Each vehicle V_i is equipped with an embedded device which provides limited computational capacity; V_i also mounts a set of sensors, each producing a stream of tuples composed by attributes $\langle y^0, \ldots, y^k \rangle$, i.e., the physical or logical time of each reading and the measurements at that time, respectively. Based on what is found in modern vehicular networks, we assume that each type of sensor produces readings with a given periodicity and that each vehicle is equipped with a storage unit that is used to maintain the sensors' readings (for the sensors deployed in the vehicle) during a given fixed period of time (e.g., during the last month). Notice that lossy data compression techniques such as PLA are not applied to the data before storing it since the allowed error bound is not known before the analyst triggers a data gathering request.

For ease of exposition, we assume in the remainder that (historical) data is stored locally at each vehicle and retrieved only when requested. We nonetheless investigate in Section A5.6 the *logical latency* incurred in a scenario in which live readings are streamed to an analysis center immediately after their compression.

We also assume that 2-way communication exists between the analysis center and each vehicle to deploy queries and to forward the sensed data, respectively. To isolate the effects of DRIVEN on data gathering and analysis from non-deterministic factors such as varying speeds and reliability of the underlying

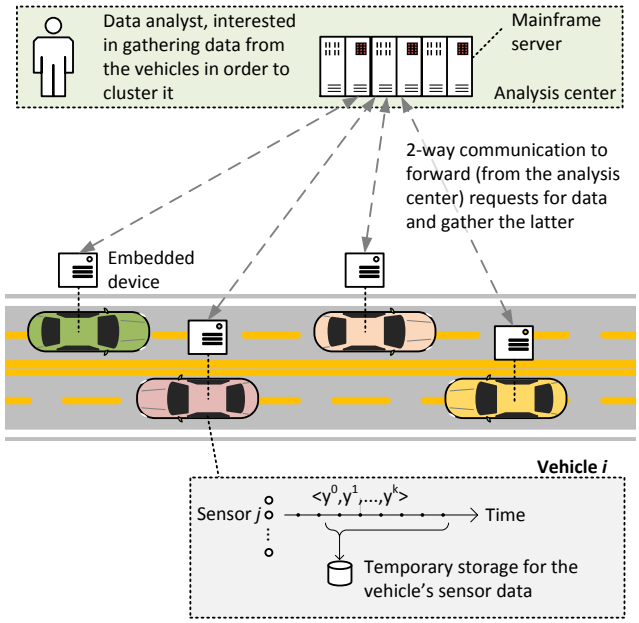


Figure A2: System model overview for DRIVEN.

communication layer, we assume this 2-way communication to provide constant upload and download speeds and no packet loss (cf. Section A5.3).

Based on the given system model illustrated in Figure A2, the goal of the DRIVEN framework is to leverage the data streaming paradigm (i.e., to define queries that gather and cluster data as DAGs of operators that can run in a distributed and parallel fashion both at the vehicles and the analysis center) while (i) only requiring analysts to provide information about the analysis' semantics (i.e., which data to gather and the distance criteria to cluster it) without composing and deploying the overall streaming query themselves and (ii) allowing for approximations in order to improve the performance (i.e., reduce the time) of retrieving the data sensed by the vehicles.

A query in the DRIVEN framework is expressed as

$$Q(\mathbb{V}, \mathbb{T}, \mathbb{S}, \Delta, [q_{\text{pre}},]\{\text{clustering parameters}\}),$$

where:

- $\mathbb{V}$ is a set of vehicles' ids,
- $\mathbb{T}$ is the period of time covered by the data to be gathered (included in the period covered by the vehicles' storage unit or referring to data being sensed live by the vehicle),
- $\mathbb{S}$ specifies the set of sensors producing the data (thus allowing the DRIVEN framework to identify the operators needed to gather the stream(s) of data they produce),

- Δ specifies the maximum error that can be introduced during the compression step while retrieving the data by the DRIVEN framework, and is further composed of $k+1$ fields, namely $\Delta_1, \Delta_2, \ldots, \Delta_k$ for a sensor with k attributes, plus Δ_0 for the (logical) time attribute,

- q_{pre} is an optional streaming query that defines pre-clustering analysis, and

- {clustering parameters} is the set of parameters used by DRIVEN's clustering component (further described in Section A4.3.2).

We refer the reader to Section A5 and Table A1 for concrete examples of queries Q and possible values of the above parameters. Notice that, being a streaming query, each DRIVEN application can be extended with additional operators to further process the found clusters (we do not discuss this since it is complementary to our work).

In order to quantify the improvement (in terms of efficiency) and the cost (in terms of precision) of the DRIVEN framework, we compare with a baseline that gathers and processes all the raw rather than the approximated data.

A4 Overview of the DRIVEN framework

In this section, we present an overview of DRIVEN. To facilitate the presentation, we first introduce a use case that serves as a running example in our discussion (we later evaluate it, together with others, in Section A5).

As discussed in Section A3, each query run by DRIVEN is a streaming continuous query deployed at both the vehicles and the analysis center, with dedicated operators for efficient data retrieval and clustering.

A4.1 Sample Use Case: Study Vehicles' Surroundings

In our running use case example, the analyst is interested in performing the clustering of LiDAR data of a bounded time interval as a preprocessing step for offboard object detection. This may help, e.g., in better understanding and improving the performance of a resource-constrained onboard object detection algorithm in a certain driving situation, and may be automatically triggered by an event such as a pedestrian crossing the road in front of the vehicle. We assume vehicles equipped with a set of LiDAR (light detection and ranging) sensors such as the ones of a Velodyne HDL-64E [155], which mounts 64 non-crossing lasers on a rotating vertical column and which, at each rotation step, shoots these lasers and produces a stream of distance readings based on the time the reflected light rays take to reach back to the sensors. Each sensor can shoot the laser 4000 times per rotation for up to 5 rotations per second, resulting thus in millions of readings per second for the whole set of sensors [142] (around 7MB/s). For each stream $\langle \alpha, \rho \rangle$ produced by one of the LiDAR sensors, the logical timestamp α allows identifying at which rotation step the distance ρ has been measured (i.e., with which angle in the horizontal

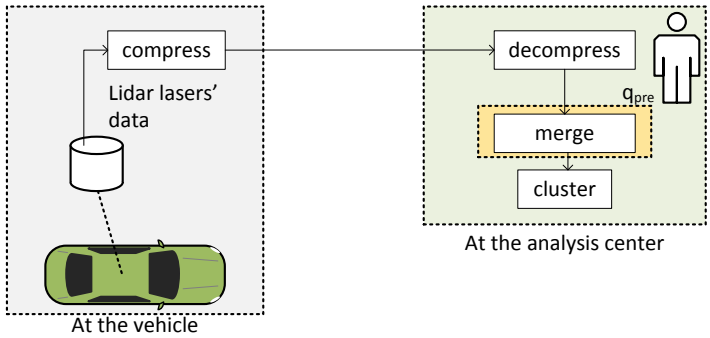


Figure A3: Overview of the modules deployed in the resulting streaming continuous query for the LiDAR use case.

plane). Notice that each reading from a sensor can be converted into a 3D point in space based on α, ρ and the elevation angle of the sensor itself.

The analyst is thus interested in the data produced over a certain period of time (e.g., covering a full rotation) by the 64 sensors mounted in each LiDAR deployed in the vehicles moving in the given urban area and relies for the clustering on a function that checks whether the Euclidean distance between any two points is within a certain threshold. Based on the query description in Section A3, the analyst could then run a query $Q(\mathbb{V}, \mathbb{T}, \mathbb{S}, \Delta, q_{\text{pre}}, \{\text{Clustering parameters}\})$ for each vehicle of interest, where:

- $\mathbb{V}$ and $\mathbb{T}$ specify from which vehicle the data should be gathered and which portion of such data should be gathered, respectively,
- $\mathbb{S}$ refers to the sets of LiDAR sensors,
- $\Delta = (\Delta_\alpha, \Delta_\rho)$ defines the maximum approximation error that is allowed when compressing the LiDAR data, bounding the rotation angle error and the distance measurement error, respectively,
- q_{pre} defines an operator merging the data from the different sensors (as further discussed in Section A5), and
- {clustering parameters} is the set of parameters later described in Section A4.3.2.

Figure A3 presents an overview of the modules deployed in the resulting streaming continuous query (each of which will be composed by one or more streaming operators, as also described in the following section).

A4.2 Data Retrieval and PLA Approximation

As discussed in Section A1, DRIVEN relies on streaming PLA to forward a compressed and lossy representation of data. To build the PLA, we use

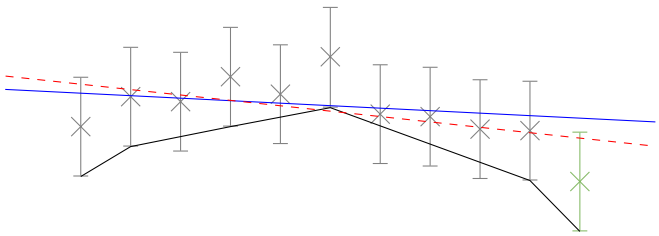


Figure A4: Best-fit lines of a set of points: solid for the first 10 points, dashed for including the 11th point (marked in green).

a construction method named *Linear*, introduced in [58], which combines several approaches of previous works on PLA such as using a best-fit line approximation [22], [81], [111] for minimizing errors and maintaining convex hulls [64], [201] for efficiently checking the violation of the error bound by the approximation. We also use here continuous processing through the *output protocol* proposed in [58] in conjunction with *Linear*, in order to balance the different trade-offs associated with PLA in streaming environments (i.e., compression ratio, reconstruction latency, and individual errors).

The *Linear* method successively updates a best-fit line through the latest not yet approximated points, until the maximum error produced by the segment approximation exceeds the tolerated error bound Δ. Updating such an estimate takes $\mathcal{O}(1)$ operations per point, but checking if the line does not violate the error condition can take up to $\mathcal{O}(n)$ if n points are currently being approximated (worst-case). However, rather than a naive sequential check that always results in the worst-case cost, by keeping track of two particular convex hulls U and L along the way (at an extra amortized $\mathcal{O}(1)$ operations per tuple), we can check the error condition in $\mathcal{O}(|U| + |L|)$ by only traversing both hulls, whose sizes are rarely higher than a few units in practice (as also observed in our extensive experimental evaluation). Figure A4 illustrates the process where input points are plotted as crosses and tolerated errors around the points are drawn as vertical line segments; the best-fit line for the first 10 points is a plain line that stays within the bounded error. By adding the 11th point (marked green), the best-fit line violates the error bound on the sixth input point (or equivalently, at the sixth input point, the approximation line is below the lower convex hull L depicted on the figure).

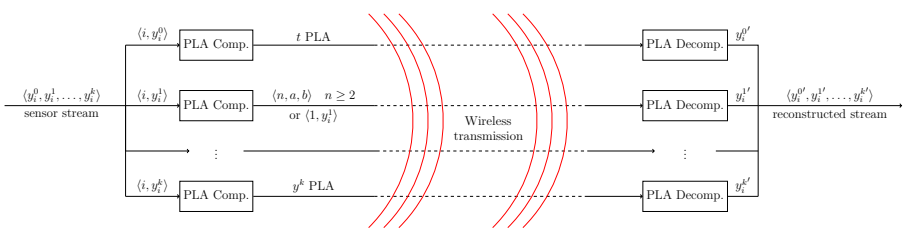


Figure A5: PLA compression / decompression flowchart with y^1's channel detailed.

Algorithm A1 PLA output logic

▷ Receive $\langle i, y_i \rangle$, while maintaining convex hulls U_{i-1}, L_{i-1} & best-fit line l_{i-1} covering $n \geq 2$ points
$l_i \leftarrow \text{newBestFitLine}(l_{i-1}, \langle i, y_i \rangle)$
$U_i, L_i \leftarrow \text{updateConvexHulls}(U_{i-1}, L_{i-1}, \langle i, y_i \rangle)$
$n \leftarrow n + 1$
if $\text{lineBreaksHulls}(l_i, U_i, L_i)$ **then**
  if $n = 3$ **then** ▷ output singleton
    output $\langle 1, y_{i-2} \rangle$
  else ▷ output segment
    output $\langle n - 1, \text{slopeOf}(l_{i-1}), \text{interceptOf}(l_{i-1}) \rangle$
else
  if $n = n_{max}$ **then** ▷ max length reached
    output $\langle n_{max}, \text{slopeOf}(l_i), \text{interceptOf}(l_i) \rangle$
  else ▷ wait for $\langle i + 1, y_{i+1} \rangle$
    continue

For the input sensor stream, composed of tuples of the form $\langle y^0, y^1, \ldots, y^k \rangle$, the different components of the PLA compression, as illustrated[(A1)] in Figure A5, are:

1. **Split:** The sensor stream is split in $k+1$ streams, one for each application-related attribute plus one additional for the timestamps. More precisely, the i-th input tuple $\langle y_i^0, y_i^1, \cdots, y_i^k \rangle$ will generate $\langle i, y_i^\ell \rangle$ on channel ℓ's stream for each $0 \leq \ell \leq k$.

2. **PLA Compression:** Each stream is compressed in parallel by computing its PLA (as depicted in Figure A1, Section A2.2) using its associated error, i.e., channel ℓ uses Δ_ℓ. Each compressor generates a PLA representation as a stream of triplets $\langle n, a, b \rangle$ or singletons $\langle 1, y \rangle$; $\langle n, a, b \rangle$ is generated for compressing n input values into a line segment whose linear coefficients are (a, b), whereas $\langle 1, y \rangle$ is generated to reproduce a single input[(A2)] of value y. In more detail, this is presented in Algorithm A1: the PLA compressor always attempts to first build the longest possible approximation segment $< n_{max}$ (of length 256, in our evaluation, cf. Section A5), but when one such segment has only length $n = 2$, thus covering only two tuples $\langle k-2, y_{k-2} \rangle$ and $\langle k-1, y_{k-1} \rangle$, $\langle k-2, y_{k-2} \rangle$ is then output as a singleton $\langle 1, y_{k-2} \rangle$. The PLA compressor continues the construction of a new approximation line segment beginning with the tuple $\langle k-1, y_{k-1} \rangle$ and the current tuple $\langle k, y_k \rangle$ (this explains the output delay associated with the singletons of Figure A1). This scheme helps to mitigate an inflation phenomenon observed when compression is low (it

[(A1)] For simplicity, the figure does not show the operators in charge of components 1 and 5.
[(A2)] Note that we add 1 in singleton tuples because both singletons and triplets are forwarded using the same channel, and thus require a prefix that distinguishes them when deserializing incoming data.

improves the compression when a single outlier tuple lies between two segment-compressible sequences of tuples).

3. **Diffusion:** The $k+1$ streams are wirelessly transmitted to the analysis center.

4. **PLA Decompression:** All streams are decompressed in parallel. The decompression algorithm is straightforward: after having already reconstructed i values on channel y^ℓ, we either generate n outputs $y'_{i+1}, \ldots, y'_{i+n}$ such that $y'_j = a \cdot j + b$ for $i+1 \leq j \leq i+n$ if the next received record is $\langle n, a, b \rangle$ on y^ℓ's transmitted PLA stream, or alternatively, we produce $y'_{i+1} = y$ if $\langle 1, y \rangle$ was received.

5. **Final Reconstruction:** The final step is to merge the $k+1$ decompressed streams to rebuild records with identical structure as the initial input stream. In particular, to reconstruct the i-th tuple $\langle y_i^{0'}, y_i^{1'}, \ldots, y_i^{k'} \rangle$ on the output stream, we need to wait for the $k+1$ decompressors to have produced at least i reconstructed values on their respective channel.

The compression scheme suggested here compresses all k attributes of a stream as well as the timestamp in the same manner (i.e., by using the tuple counter i for each tuple $\langle i, y_i^\ell \rangle$ on a sensor attribute y^ℓ). This differs from the scheme suggested in [90], where a counter was used only for the timestamp stream $\langle i, y_i^0 \rangle$ while the remaining attributes of the stream were compressed using the original timestamps, and decompressed using the reconstructed timestamps. As explained in [90], this scheme could not guarantee a bounded reconstruction error at all times as errors from timestamp reconstruction could propagate to the reconstruction of other channels. The new scheme proposed here results in similar compression and performance figures on our evaluated data, as discussed later in Section A5, and guarantees a bounded reconstruction error.

A4.3 Data Clustering with Lisco

As described in Section A2, distance-based clustering approaches form clusters using a given distance metric. Since computing the distances of one tuple from all the other tuples in a certain dataset in order to find the ones within a threshold distance would incur an $\mathcal{O}(n^2)$ complexity when running all-to-all comparisons, it is necessary to prune the search space. For this purpose, several clustering approaches have an intermediate step after data acquisition and before the main clustering algorithm. This additional step builds an extra supporting data structure (e.g., a kd-tree [66]) in order to organize the collected data before performing the clustering. In this way, a batch-based processing is introduced which results in an average $\mathcal{O}(n \log n)$ cost [193] but requires multiple passes over the data (possibly affecting the performance).

Lisco is a recently proposed method that overcomes the batch processing disadvantages through a single-pass continuous distance-based clustering (Euclidean in the original paper [142]) that exploits inherent orderings of data

(when such orderings are present). The intuition behind Lisco is to store the data in a simpler data structure that preserves such inherent ordering and therefore eliminates the need for an extra supporting sorting data structure. In the original paper, it is discussed (and empirically observed) that storing and organizing data tuples using Lisco have $\mathcal{O}(1)$ complexity and can be performed during the data acquisition step which results in an average $\mathcal{O}(n)$ cost.

While we rely on the LiDAR-based use case to overview DRIVEN and its clustering component, our implementation of Lisco within DRIVEN opens up for the clustering of other data too, as we discuss in the following.

A4.3.1 Clustering LiDAR Data (Intuition)

Figure A6 a) shows Lisco's intuition for clustering LiDAR data. As shown, for a certain point p hit by a laser, the search for neighbors within a certain (Euclidean) distance can be limited to a certain set of lasers and angles (based on p's distance and angles). The *neighbor mask*, containing possible points hit by such lasers (and for the given angles), specifies the portion of data outside of which neighbors can *not* be found for p. This limits the search space for

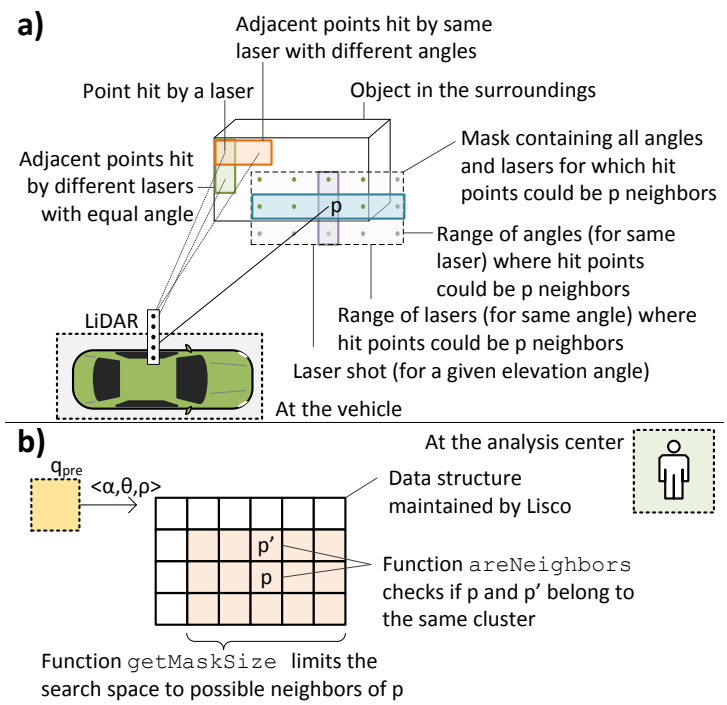


Figure A6: Example of how the search space for a point p (for the LiDAR use case) can be limited to points potentially reported by lasers (and with certain angles) within a mask centered in p (a) and the corresponding 2D matrix maintained by Lisco (b).

p's neighbors to the points measured for the given range of angles and lasers. Notice that such points must be checked since not all angles and lasers falling within the given ranges necessarily hit a point that is a neighbor of p, as shown in the figure. Internally, Lisco can then maintain incoming points in a 2D array.

A4.3.2 Lisco Generalization in DRIVEN

The Lisco implementation in DRIVEN maintains data in an n-dimensional array and clusters incoming tuples while they are stored in it. One of the n dimensions is given by the y^0 attribute while the other *optional* $n-1$ dimensions can be specified as attributes of the tuple's schema. In this way, the analyst can leverage any implicit sorting carried by one or more attributes of the tuples produced by q_{pre} (aside from the timestamp itself) to speed-up Lisco's clustering. To do this, the first clustering parameter defined by the analyst is an optional list of attributes to define the additional $n-1$ dimensions of Lisco's internal multi-dimensional array. It should be noticed that, for each attribute y^k specified as a dimension by the analyst, the latter must also specify the range of values observable for it, for DRIVEN to setup Lisco's internal data structure.

The second and third clustering parameters are the functions `int[n] getMaskSize(Tuple` τ`)` and `boolean areNeighb(Tuple` τ_1`,Tuple` τ_2`)`. The former function specifies how far (in the sense of indexes) Lisco should explore any of the n dimensions of the array around tuple τ, to search for potential candidates for clustering. Lisco employs the return values of this function to create the neighbor mask and bound the search space around τ. Internally, Lisco runs the aggregation over any of the n dimensions as soon as the latter is filled for a given value (e.g., when all the tuples sharing the same y^0 values are received). The latter function is used to check if two tuples falling into the same neighbor mask should be clustered together or not.

Finally, the analyst must also specify the minimum number of points $minPts$ to differentiate clusters from noise (Definition A1).

Continuing the example in Figure A6 b), the schema of the tuples produced by q_{pre} could in this case carry attributes $\langle \alpha, \theta, \rho \rangle$, where α is the logical timestamp that refers to a certain angle of the LiDAR sensor, θ is the elevation angle (based on the laser producing the reading) and ρ is the measured distance. To store the tuples, Lisco could be instructed to keep data in a 2D matrix where consecutive readings from the same laser are assigned to columns and the different lasers are assigned to different rows (as done in [142]). In this way, the ordering of two dimensions of the tuples would be kept. Using the 2D matrix to store the data, `int[n] getMaskSize(Tuple` τ`)` can be implemented to return the neighbor mask of a tuple τ in terms of a limited number of rows and columns around τ. Finally, the `areNeighb(Tuple` τ_1`,Tuple` τ_2`)` can be implemented to check whether the Euclidean distance between the readings of tuples τ_1 and τ_2 is within the threshold defined by the analyst.

A5 Evaluation

We evaluate in here the tradeoffs in compression, approximation error, retrieval time and clustering quality for DRIVEN. We first present the datasets used, the software and hardware setup and then discuss four different use cases in which historic data is gathered and clustered. Finally, we gauge PLA's compression performance by comparing it to a lossless, general-purpose compression technique (ZIP) and discuss the concept of inherent logical latency of PLA compression to investigate the impact of DRIVEN on queries gathering live rather than historic data.

A5.1 Data

We use three datasets in our evaluation.

1. The *Ford Campus* dataset [155], providing data generated by a VelodyneHDL64E roof-mounted LiDAR (see Section A4.1 for an overview of LiDAR) from one vehicle. Each file in the dataset corresponds to one full rotation of the LiDAR, which consists of 64 individual lasers mounted in a column. According to our system model (Section A3), each of the 64 lasers is an individual sensor, with the sensor ID being the sensor's fixed elevation angle. In our implementation, each laser stores its data in a dedicated file.

2. The *GeoLife* dataset, containing GPS data collected in the *GeoLife* project by 182 users over ca. three years [213]–[215]. We use a subset of the dataset with vehicular GPS traces in the Beijing area.

3. The *Volvo* dataset, provided by Volvo Cars. This dataset consists of CAN data[(A3)] and GPS traces from 20 hybrid cars and was collected in Sweden in the years 2014 and 2015.

A5.2 Software and Hardware Setup

We implemented Lisco in Python 3.6 and the PLA components (see Section A4.2) in Apache Flink 1.5.0. The segment length n of the PLA compressor is bounded by 256 to limit its bandwidth consumption to 1 byte, while the parameter *minPts* is set to 10 in all experiments (this parameter is adopted from [142]).

We use as a stand-in for the vehicle node an ODROID-XU3 single-board computer to approximate the low-power processor of a vehicle, equipped with a Samsung Exynos 5422 Cortex-A15 2.0GHz quad-core and Cortex-A7 quad-core CPU and 2 GB of LPDDR3 RAM at 933MHz. For the analysis center, we use a server with an Intel(R)
Core(TM) i7-4790 3.60GHz quad-core CPU and 8GB of RAM. The ODROID and the server are connected via Ethernet. We reduce the connection bandwidth

[(A3)] CAN (Controller Area Network) is a vehicular communication bus standard over which sensor data, fault messages, etc. can be transmitted.

using the tool *trickle* [65] to simulate four different upload speeds for the ODROID: *slow* (8KB/s, in the range of 2G), *medium* (500KB/s, in the range of 3G), *fast* (1000KB/s, in the range of 4G) and *very fast* (10000KB/s, in the range of 5G).

A5.3 Evaluation Metrics

DRIVEN is evaluated for four use cases among four dimensions:

- *Average error*: The average error $\bar{Y} = \frac{1}{N}\sum_{i=1}^{N}|y_i - y'_i|$ between original values y_i and reconstructed ones y'_i.
- *Compression ratio*: The size of the compressed data divided by the raw data size.
- *Adjusted rand index*: The clustering obtained from the approximated data compared to the clustering obtained from the original data via the adjusted rand index.[(A4)]
- *Gathering time ratio*: The time needed to gather the approximated data (including compression / decompression overheads) divided by that taken to gather the raw data.

For all use cases, we use the term *baseline* to refer to a setup in which raw data (i.e, with no compression) is gathered and clustered.
In addition to the four evaluation dimensions explained above, we also evaluate the *Logical Latency* for the *Ford Campus* and *GeoLife* datasets (we refer the reader to Section A2.4 for a definition of logical latency).
Since simulating the communication behavior of a large vehicular network is beyond the scope of this work (and based also on the observation that real behavior would depend on factors we cannot predict, such as the position of a certain vehicle), experiments studying the gathering time are set up to favor the baseline over DRIVEN and thus avoid bias. More concretely, gathering time is measured for the collection of each sensor's data without concurrent or parallel transfers from multiple vehicles, thus avoiding overheads (e.g., packet losses) proportional to the size of the transmitted information (i.e., higher for the baseline, given that raw data is larger in size than the compressed one, as we show in the following).

In the following, results are presented through violin plots. Violin plots show the distribution of the underlying data along their vertical axis, with the mean marked by a horizontal bar. All the presented plots contain data from at least 55 experiment runs.

[(A4)]The rand index of two partitions (or sets of clusters) A, B is a symmetric measure that counts how many pairs of elements in partition B are clustered exactly as in partition A. The adjusted rand index extension takes into account accidental random clusterings (see [192] for more details).

Table A1: Queries $Q(\mathbb{V}, \mathbb{T}, \mathbb{S}, \Delta, q_{\text{pre}}, \{\text{clustering parameters}\})$ for evaluation. See Section A3 for explanation of query arguments.

Q_1: **LiDAR**

$\mathbb{V}$	$\mathbb{T}$	$\mathbb{S}$	Δ	q_{pre}	clustering parameters

1 vehicle	10 rot.	LiDAR 64 lasers	$\Delta_\alpha = 0.0005$rad, $\Delta_\rho \in [0.1, 0.2, 0.5, 1, 5, 10]$ m	merge 64 lasers
				add laser id
					`getMaskSize(Tuple` τ`):`
					`return getMaskSizeInRot(`τ`)`
					`areNeighb(Tuple` τ_1`, Tuple` τ_2`):`
					`return euclidDist(`τ_1`,`τ_2`)≤0.5m`

Q_2: **1-Vehicle 1-Day**

$\mathbb{V}$	$\mathbb{T}$	$\mathbb{S}$	Δ	q_{pre}	clustering parameters

1 vehicle	1 day	GPS	$\Delta_t = 1$ s, $\Delta_x, \Delta_y \in [1, 2, 5, 10, 20, 50]$ m	`windowAggr(5s)`
				emit latest tuple
					`getMaskSize(Tuple` τ`):`
					`return 12`
					`areNeighb(Tuple` τ_1`, Tuple` τ_2`):`
					`return euclidDist(`τ_1`,`τ_2`)≤50m`

Q_3: **1-Vehicle 14-Day**

$\mathbb{V}$	$\mathbb{T}$	$\mathbb{S}$	Δ	q_{pre}	clustering parameters

1 vehicle	14 days	GPS	$\Delta_t = 1$ s, $\Delta_x, \Delta_y \in [1, 2, 5, 10, 20, 50]$ m	`windowAggr(10s)`
				add day id
				emit latest tuple
					`getMaskSize(Tuple` τ`):`
					`return 15, 14`
					`areNeighb(Tuple` τ_1`, Tuple` τ_2`):`
					`return euclidDist(`τ_1`,`τ_2`)≤100m`

Q_4: **Car usage grids**

$\mathbb{V}$	$\mathbb{T}$	$\mathbb{S}$	Δ	q_{pre}	clustering parameters

20 vehicles	7 days	GPS
		electric RPM
		comb. RPM
			$\Delta_t{}^{\text{G}} = 1$ s, $\Delta_t{}^{\text{e}} = \Delta_t{}^{\text{c}} = 0.005$ s
			$\Delta_x, \Delta_y \in [1, 2, 5, 10, 20, 50]$ m
			$\Delta_\omega{}^{\text{e}}, \Delta_\omega{}^{\text{c}} \in [1, 5, 10, 20, 50, 100]$ Hz
				add day id
				find drive mode
				add to grid
					`getMaskSize(Tuple` τ`):`
					`return 7, 2, 2`
					`areNeighb(Tuple` τ_1`, Tuple` τ_2`):`
					`return counterDist(`τ_1`,`τ_2`)≤1)`

A5.4 Use Cases

A5.4.1 Q_1 LiDAR

This is the use case presented in detail in Section A4.1. In accordance with our system model, the data for each of the 64 lasers is stored on-vehicle as a stream of $\langle \alpha, \rho \rangle$ with the azimuth angle α (logical timestamp) and the distance reading ρ. The query for this use case is detailed in Table A1. Based on the query, all the sensor reading streams from the last ten rotations from each of the 64

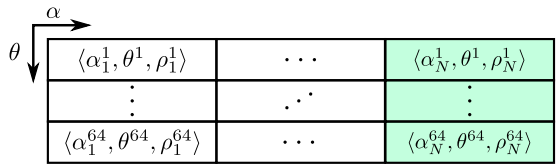


Figure A7: Q_1: Sketch of data structure produced by q_{pre}.

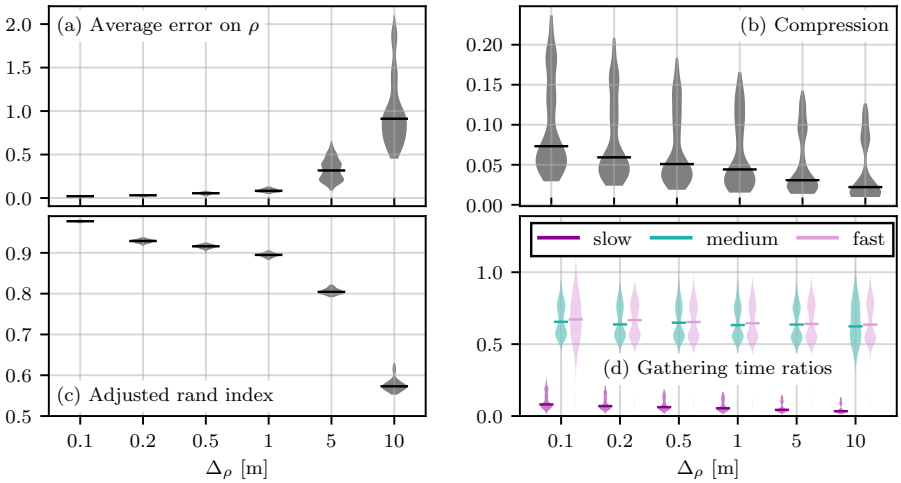


Figure A8: Q_1: (a) - (c) Compression and clustering statistics for various Δ_ρ; (d) gathering time ratio for various Δ_ρ and different network speeds.

LiDAR lasers from one vehicle are successively compressed on-vehicle with some maximum errors $\Delta_\alpha, \Delta_\rho$ on the logical timestamps α and the distance readings ρ. Each compressed stream of laser readings is then successively sent to the analysis center, where the streams are decompressed. Query q_{pre} assigns each tuple its laser id and the horizontal angle θ, providing to Lisco the data structured as the 2D matrix (Figure A7) to Lisco. The tuples are added column-wise (see colored column) by q_{pre} with decreasing laser id θ^i (the id of the i-th laser) from top to bottom and increasing rotation angle α^i_j (the j-th rotation step of the i-th laser) from left to right. This merging of data from different sensors is performed deterministically [84] based on the logical timestamp α carried by the tuples.

As clustering parameters, Lisco is instructed to check the Euclidean distance between pairs of tuples; to accomplish that, it searches for candidates in a maximum α, θ - area around tuple τ defined by `getMaskSizeInRot(`τ`)`, which calculates the angles α, θ of the horizontal and vertical laser beams who could hit points that are within distance $\epsilon = 0.5$m from the sensor reading corresponding to τ, as they bound the ϵ-neighborhood of the latter, while also ensuring that only points within the same rotation are part of the mask.

Through the way that data is maintained in the 2D matrix, this defines a rectangular area (i.e., mask) in the matrix around τ (cp. Section A4.3.1).

The compression statistics for this use case can be seen in Figure A8 (a) and (b) expressed via violin plots. The angle α is in all cases compressed with a maximum error Δ_α of 1.5×10^{-3}rad, yielding an average error on α of $3.4 \times 10^{-5} \pm 1.0 \times 10^{-5}$rad (average $\pm$ standard deviation). ρ is compressed for values $[0.1, 0.2, 0.5, 1, 5, 10]$m. In (a), it appears for larger values of Δ_ρ that the average error is about one order of magnitude smaller than the maximum error (this is true for small Δ_ρ too, although harder to appreciate in the graph). The compression as a ratio of compressed vs. raw file size in (b) shows that LiDAR data can already for a maximum error $\Delta_\rho = 0.1$m be compressed below (median) 10% of the raw size, which we attribute to the regularity of the logical timestamps α as well as to the existence of stretches of $\rho = 0$ in the raw data (these occur when the laser is not reflected, cp. Section A4.1). For increasing maximum error, the compression ratio decreases only slightly, which indicates that almost maximum compression is reached early. The long tails towards lower compression hint at single files with data that is harder to compress than the average. For $\Delta_\rho = 1$m, the data transmitted in this use case (10 rotations of the LiDAR, which are completed in 2s) is around 750KB. Transmitting this live is possible with network speeds of 3G or larger. The comparison of the resulting clusters from query Q_1 with the baseline is shown in Figure A8 (c) (as only points within the same rotation may be clustered, we compare the resulting clusters between the same rotations, not between the sets of ten rotations). One observes that for increasing Δ_ρ the similarity between the clusters of approximated and baseline data decreases. However, for $\Delta_\rho = 0.1$m, the median compression ratio is already below 0.10, while the median adjusted rand index is larger than 0.95, indicating that a large compression can be achieved without a large loss of precision in the clustering.

Figure A8 (d) shows the end-to-end gathering time ratio for the three network speeds. While there is no significant decrease for increasing maximum error (according to the compression ratio that also decreases only slightly for larger Δ_ρ), for all network speeds data gathering times are reduced to between 60% for fast networks down to less than 15% for slow networks.

A5.4.2 Q_2 1-Vehicle 1-Day

In this use case, the analyst requests the GPS data of a single day from one vehicle in order to cluster all points within a predefined distance and timespan. This could, e.g., serve to identify areas of slow traffic or areas where the vehicle stopped. Based on our system model, the data is stored on-vehicle as a stream of tuples $\langle t, x, y \rangle$ with the timestamp as the actual measurement time and the x, y attributes being the coordinates in meters.

The query Q_2 for this use case is described in detail in Table A1. The

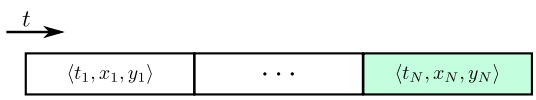


Figure A9: Q_2: Sketch of data structure produced by q_{pre}.

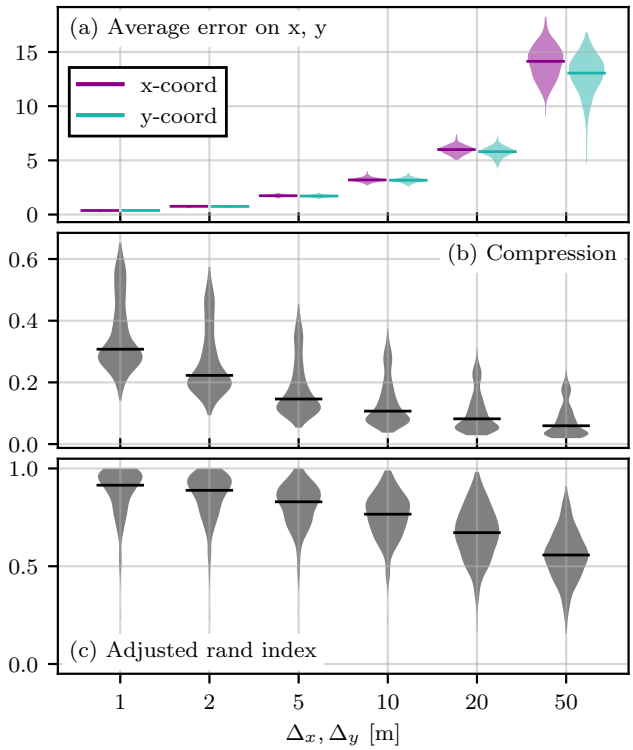


Figure A10: Q_2: (a), (b) Compression statistics; (c): Adjusted rand index.

GPS position data stream from one day and one specific vehicle is compressed on-vehicle with some error Δ_t on the timestamps and errors $\Delta_x = \Delta_y$ on the vehicle's GPS coordinates and sent to the analysis center. There, the stream of decompressed tuples is aggregated by q_{pre} in tumbling windows of 5 seconds, and for each window, only the latest tuple is returned as soon as the window completes. The data provided to Lisco structured as a 2D matrix is sketched in Figure A9 (the colored field contains the last added tuple). If a window is empty because no data exists for the corresponding time period, the field in the data structure will also remain empty.

As clustering parameters, Lisco is instructed via `getMaskSize(`τ`)` to check the last 6 indexes of the 1D array, reducing the search space for neighbors and ensuring the clustering only of points that are also close in time. The clustering decision is taken by the function `areNeighb(`τ_1`,`τ_2`)` on the basis of the Euclidean distance between the x, y-coordinates of the two tuples.

The compression statistics are given in Figure A10 (a), (b). We choose in this case a fixed error $\Delta_t = 1$ s for the compression of the timestamps, resulting in an average error of (0.095 ± 0.090) s. Δ_x and Δ_y are chosen to be equal and $\in [1, 2, 5, 10, 20, 50]$ m. Figure A10 (a) shows the average error on both coordinates as a function of the maximum errors, with the average errors being roughly one-third of the allowed maximum error. Figure A10 (b) shows the

total compression achieved for each maximum error: assuming a measurement uncertainty for GPS data on the order of few meters, maximum compression errors of less than ten meters may be assumed to be small. Still, these result in compression ratios that can be lower than 0.2. This may be explained with straight roads, resulting in long, linear segments in the GPS data, as well as regularity of the timestamps. The violin plots' long upward tails hint at individual files with lower compressibility. The results for the comparison of the resulting clusters from approximated and raw data are shown in Figure A10 (c): for small maximum errors Δ_x, Δ_y, the adjusted rand index is close to 1, but it decreases for larger maximum errors.

The gathering time ratios are shown in Figure A11 (a). The median is around 1.3 for faster networks and does not decrease for increasing compression. This shows that DRIVEN in this use case is only beneficial for a slow network. More insight is gained from Figure A11 (b), showing the gathering time ratios as a function of the baseline data size for a medium speed network. For small data sizes, the additional time overhead of the compression and decompression procedure increases the gathering duration over directly transmitting the raw sample. For larger sample sizes, the gathering time ratio approaches 1 for all maximum errors Δ_x, Δ_y. This gives an approximation for the minimum size of data to be collected given the network bandwidth and the compression/decompression overheads, as we further show in the remainder (we stress nonetheless that our evaluation setup favors raw data gathering, as explained in Section A5.3).

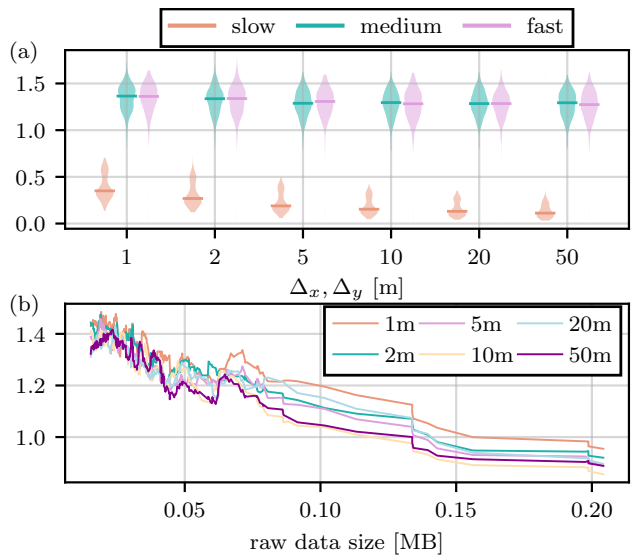


Figure A11: Q_2: Gathering time ratios for various (a) maximal errors for different network speeds and (b) raw data sizes (rolling average over 13 values, different colors are used for distinct values of Δ_x, Δ_y) for a medium speed network.

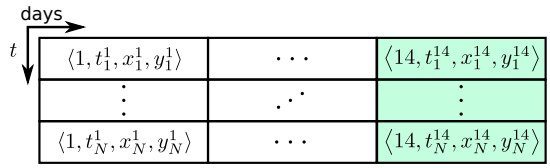


Figure A12: Q_3: Sketch of data structure produced by q_{pre}.

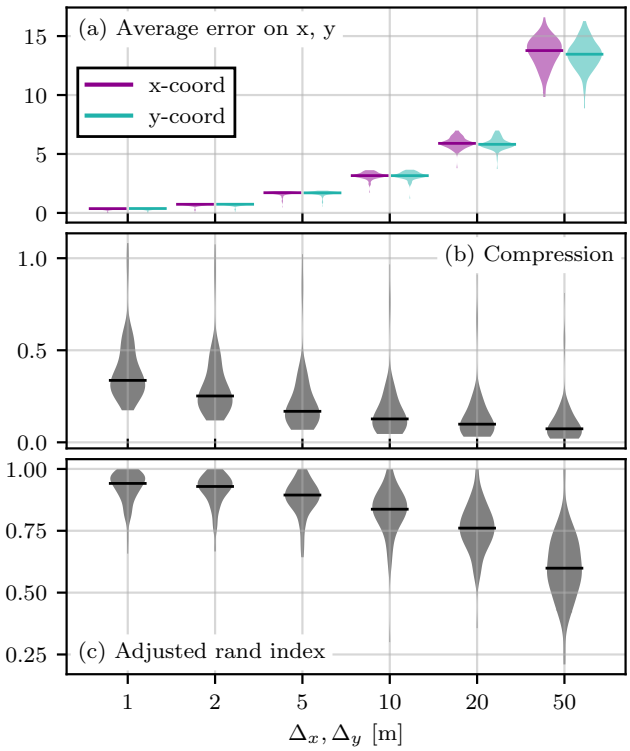


Figure A13: Q_3: (a), (b) Compression statistics; (c) Adjusted rand index.

A5.4.3 Q_3 1-Vehicle 14-Day

In this use case, the analyst requests the GPS data from one specific vehicle from the last 14 days, to possibly identify routes that one vehicle follows regularly. The query Q_3, described in the query overview in Table A1, differs from Q_2 in the period covered by the data and in the task of q_{pre}: upon consecutively receiving the GPS streams for each day and windowing (with 10 second windows) as in the previous use case, each tuple is also assigned an identifier for the day. As soon as the stream for one day is processed, it is added column-wise to a data structure as shown in Figure A12 (the first entry of each tuple is the day identifier id, t is the number of seconds from midnight on day id). Lisco can thus process the GPS stream of each day as soon as it is received. In contrast to the previous use case, Lisco is now instructed to search

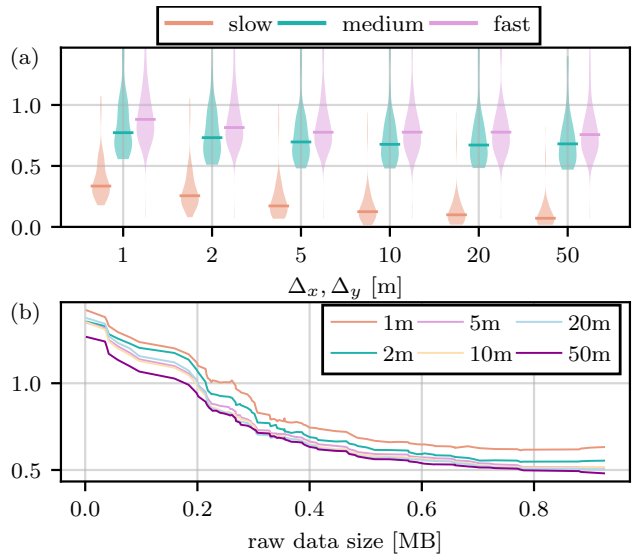


Figure A14: Q_3: Gathering time ratios for various (a) maximal errors for different network speeds and (b) raw data sizes (rolling average over 13 values, different colors are used for distinct values of Δ_x, Δ_y) for a medium speed network.

the last 15 cells in the direction t, and all cells in the direction "days" (14 ensures that all days are encompassed), for tuples within a Euclidean distance of 150 m.

The compression statistics may be found in Figure A13 (a), (b) and are similar to those seen in Figure A10 (a), (b), as the same data type with only increased sample size is used. Here the constant maximum error $\Delta_t = 1$ s results in an average error of (0.093 ± 0.081)s. The evaluation of clustering qualities is shown in Figure A13 (c), also with similar results to the previous use case. The addition of the attribute "days" for Lisco seemingly has only a small influence, suggesting that in the majority of samples there is no significant number of inter-day clusters.

Figure A14 (a) shows the measured gathering time ratios. The median gathering time ratios are below 0.35 for all values of the maximum errors for a slow network, and for faster networks around 0.75, although also samples with ratios greater than 1 are present. As shown in Figure A14 (b) (for medium network speeds), this is due to samples with raw data size smaller than 200KB (at medium network speeds). For samples larger than 200KB, gathering time ratios for all values of Δ_x, Δ_y are smaller than 1.

A5.4.4 Q_4 Car Usage Grids

In this use case, the analyst wants to investigate if a fleet of hybrid cars uses the same drive mode (electric/traditional) on the same routes at similar times of the day. To perform this query, the analyst requests GPS data as well as the combustion engine and electric rear axle engine (ERAD) RPMs (rotation per

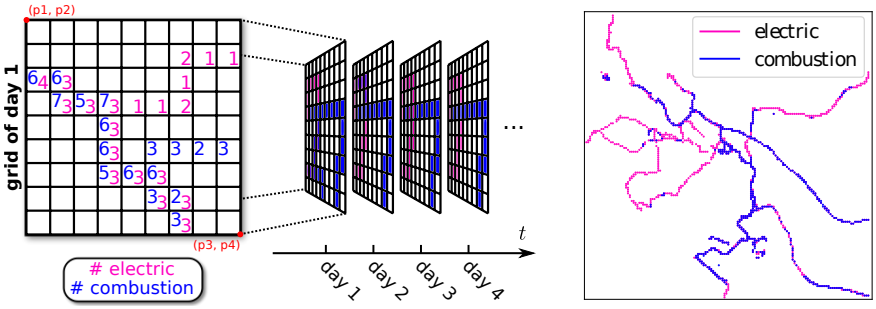


Figure A15: Data structure of Q_4. (a): Sketch of data structure produced by q_{pre}. (b): Example of one grid (instead of showing the number of vehicles per drive mode and cell, only colors indicate the cell occupation).

minute) time series, requiring three different time series from three different sensors, for one week from 20 hybrid cars. The three time series in tuple notation are $\langle t^{\text{G}}, x, y\rangle$ for GPS (physical timestamp [s], x-coordinate [m] and y-coordinate [m]), $\langle t^{\text{e}}, \omega^{\text{e}}\rangle$ for the ERAD RPM (physical timestamp [s], RPM [Hz]) and $\langle t^{\text{c}}, \omega^{\text{c}}\rangle$ for the combustion engine RPM (physical timestamp [s], RPM [Hz]). Using this data, a map is created for each day, and clusters of identical drive mode (electric/combustion engine use) between different days and different locations on the map are created to identify routes for which a certain drive mode is preferred.

Over a rectangular geographic grid of 150×150 cells, the GPS trace of a car V_i during each day of the requested week is discretized. For each cell, characterized by the time period T during which V_i was present within the cell's boundaries, the combustion and electric engine RPMs during T are regarded and a decision is taken whether the car was in combustion or electric mode during T. Each cell contains a counter for each mode, and if V_i is found to be in a certain mode while in that cell then the corresponding mode counter is increased. For each day, each V_i can only contribute to each cell's counter once. This is repeated for all $V_i, i \in \{1, \ldots, 20\}$, such that a map is created for each day of the week containing the cells visited (including drive mode) by all vehicles.

This pre-processing task is performed by q_{pre}, and a sketch of the data structured as a 3D matrix that is passed to Lisco is visualized in Figure A15(a): The coordinates (p1,p2) and (p3,p4) are located at the corners of the geographical grid (which in this sketch is a 9×9 grid). The first dimension of the 3D matrix is time (day of the week), the other two are geographic x, y-coordinates. A concrete example grid for one day is shown in Figure A15(b) (for visibility, the counters for each cell are represented with only a color marker).

The query Q_4 is formally described in the query overview in Table A1. Two grid cells, represented by tuples τ_1, τ_2, become clustered in accordance with `counterDist(`τ_1, τ_2`)` if the difference in both cells' electric or both cells' combustion mode counter is smaller than or equal to 1.

#	$[\Delta_x = \Delta_y, \Delta_{\omega^e}, \Delta_{\omega^c}]$
1	[1 m, 1 Hz, 1 Hz]
2	[2 m, 5 Hz, 5 Hz]
3	[5 m, 10 Hz, 10 Hz]
4	[10 m, 20 Hz, 20 Hz]
5	[20 m, 50 Hz, 50 Hz]
6	[50 m, 100 Hz, 100 Hz]

Table A2: Q_4: Maximum-error sets used. As three different time series are requested, three different error parameters are given (for GPS, $\Delta_x = \Delta_y$).

The time channels $t^{\mathrm{G}}, t^{\mathrm{e}}, t^{\mathrm{c}}$ (for GPS, electric engine RPM and combustion engine RPM) are compressed with $\Delta_{t^{\mathrm{G}}} = 1$ s; $\Delta_{t^{\mathrm{e}}} = \Delta_{t^{\mathrm{c}}} = 0.005$Hz, resulting in average errors of (0.49 ± 0.06)s, $(142 \pm 72)\mu$s and $(921 \pm 161)\mu$s, respectively. The remaining channels are compressed for six sets of maximum errors shown in Table A2.

We assume for the evaluation that there is no change in network and analysis center performance for gathering the data from 20 vehicles at once, and thus

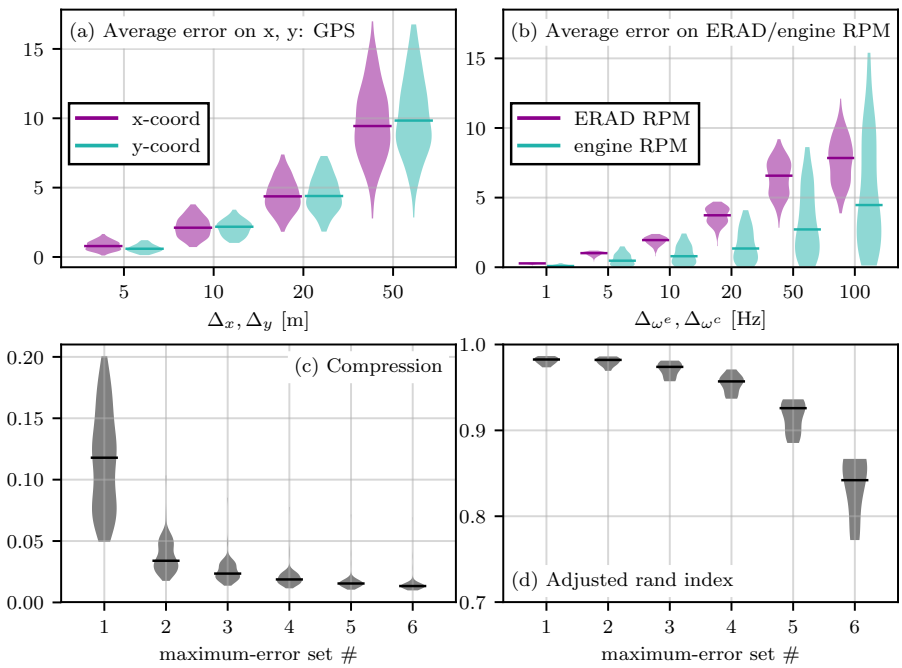


Figure A16: Q_4: (a) Average error on the x- and y-coordinate for several values of the maximum compression errors Δ_x, Δ_y for GPS data; (b) average error on the ERAD and engine RPM for several values of the maximum compression errors $\Delta_{\omega^e}, \Delta_{\omega^c}$; (c) compression statistics for different maximum-error sets (see Table A2); (d) adjusted rand index.

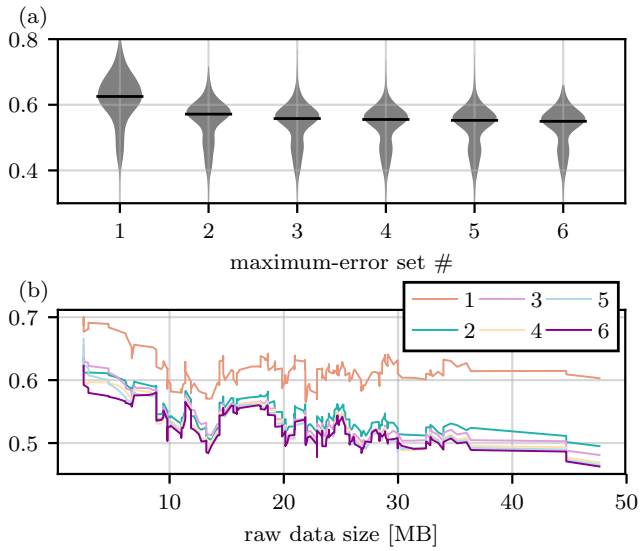


Figure A17: Q_4: Gathering time ratios for (a) a very fast network speed and (b) for various raw data sizes (rolling average over 13 values, different colors are used for different maximum-error sets) for a very fast network.

simulate the query on one vehicle only (i.e., utilizing one ODROID as in the other use cases).

The compression and clustering statistics for this use case are shown in Figure A16: (a) is the average error on the x, y-coordinate of the GPS time series for different values of $\Delta_x = \Delta_y$. The average errors for the first two error sets are not displayed, due to the low geospatial precision of the GPS time series the average error on the GPS coordinates is on the order of 10^{-5}m for $\Delta_x = \Delta_y \in \{1\text{m}, 2\text{m}\}$. (b) is the average error on both the ERAD and the combustion engine RPM, which are roughly one order of magnitude smaller than the allowed maximum errors $\Delta_{\omega^e}, \Delta_{\omega^c}$. (c) shows that already for the maximum-error set #1 a median compression of 12 % can be achieved, down to 2% for #6. This is explained by long stretches of inactivity of either the ERAD or the combustion engine, resulting in long stretches of constant zero readings in their respective time series. These stretches can be compressed well with PLA. The adjusted rand indices in (d) remain in the median above 0.9 until maximum-error set #6, indicating that the analysis accuracy in this use case is quite robust towards compression.

Gathering time ratios for a very fast network are shown in Figure A17 (a) for the six different maximum-error sets. For the smallest individual maximum errors at maximum-error set #1, the gathering time ratio is below 0.65, and for increasing individual maximum errors the gathering time ratio decreases slightly more to 0.55, but is almost constant. This may be due to the almost-constant compression for higher maximum-error sets, see Figure A16 (c). Figure A17 (b) shows the gathering time ratios for the same very fast network speed for various raw data sizes. For all maximum-error sets, the gathering time ratio

tends to decrease for increasing raw data size. The noisy behavior of the curves, which is almost identical for each maximum-error set, may hint at individual files that are harder or easier to compress than other files of similar raw data size.

A5.5 Compression Evaluation

To gauge the performance of our PLA compression technique, we compare it with the `DEFLATE` compression algorithm used for ZIP compression. We choose ZIP because of its general-purpose nature, widespread use and lossless compression. Here, we show a comparison for the data used in Q_1 (LiDAR) and Q_2 (GPS). In our experiments, we zip for each separate channel n_{zip} consecutive points (thus $n_{zip} \cdot \text{size}(float)$ bytes) and take the average over all channels per file. The results are plotted in Figure A18, with "$\times$" marking the compression achieved with PLA plotted in n_{zip}'s column corresponding to the average segment length obtained through DRIVEN (be reminded that the segment length with our PLA method varies and depends on the underlying data; only the maximum segment length is specified and set to 256 points). We set here the maximum tolerated errors to minimal-loss values, i.e., $\Delta_\rho = 0.01$m for LiDAR, and $\Delta_x = \Delta_y = 1$m for GPS, cf. Figures A8 (a), A10 (a). For comparable segment lengths, the ZIP representation is 2-10 times larger, indicating the advantages of lossy, piecewise linear compression for this type of data and scenario. Even when zipping all the available data ($n_{zip} = \infty$), the gap remains stark, which further hints at the validity of our PLA implementation. Moreover, allowing for larger segment lengths would limit the usefulness in a live data gathering scenario, as shown in the following subsection.

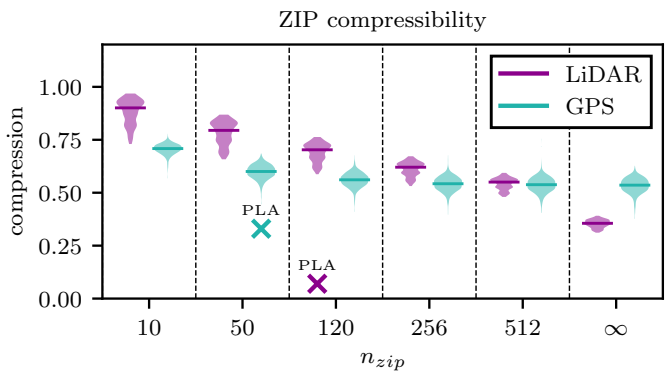


Figure A18: Compression ratios using ZIP for varying segment lengths n_{zip}. "$\times$" marks the compression achieved with PLA for equivalent *average* n for smallest maximum errors (almost lossless).

A5.6 Logical Latency

Lastly, we evaluate DRIVEN by studying the logical latency observed when compressing data from the *Ford Campus* and *GeoLife* datasets. By doing this, we can thus estimate the live gathering time incurred when performing PLA compression on live data (which can be approximated by the average segment length multiplied with the sampling period of data being clustered, since this is orders of magnitude larger than emission time, as well as transmission and reconstruction delays).

Notice that we do not present results for the *Volvo* dataset in this case, since the different order of magnitude of sampling period between GPS and ERAD/engine data (seconds versus milliseconds) results in GPS data (already discussed for the *GeoLife* dataset) being the one dominating the live gathering time delay for compression of live sensed data. More concretely, given the GPS' data sampling period of 5 s and the ERAD/engine RPMs sampling period of 40 ms, the shortest possible segment of GPS' data (approximating 3 points) results in an average live gathering time of 10 seconds while the longest possible segment of ERAD/engine RPMs data (approximating 256 points) results in an average live gathering time of approximately 5 seconds.

Based on our description of logical latency (Section A2.4), the logical latency incurred by DRIVEN can be modeled as follows. For the stream of values y, the logical latency is obtained as the difference $j - i$ where $\langle j, y_j \rangle$ is the last tuple read before the compressor sends information that triggers the i-th tuple's reconstruction on the decompressor side; two situations are then possible: either processing $\langle j, y_j \rangle$ triggers the emission of a line segment $\langle n, a, b \rangle$ and in this case $\langle i, y'_i \rangle$, with $j - n - 1 \leq i \leq j - 1$, is reconstructed among $n - 1$ other tuples using the segment's information, or $\langle j, y_j \rangle$ triggers the emission of a singleton $\langle 1, y'_i \rangle$ where then $i = j - 2$ (since 3 tuples are read before emitting a singleton, the logical latency is always 2 in this case). Logical latencies are bounded by the maximum segment length (this occurs for the first tuple on a maximum-length segment), and the average logical latency corresponds (when omitting singletons) to half the average segment length. When no compression is performed, logical latencies are 0. For an input tuple $\langle y_i^0, \ldots, y_i^k \rangle$ (which is split into $\langle i, y_i^0 \rangle, \ldots, \langle i, y_i^k \rangle$), the *combined* logical latency is the maximum logical latency of the individual tuples $\langle i, y_i^0 \rangle, \ldots, \langle i, y_i^k \rangle$, as the original tuple can only be reconstructed as soon as all its attributes have been individually reconstructed. The compression scheme used in DRIVEN, i.e., the PLA construction method *Linear* coupled with a streaming-based protocol, has been shown in [58] to produce logical latencies one to two orders of magnitude smaller than other state-of-the-art PLA construction algorithms.

In addition to calculating the average logical latencies of the *Ford Campus* and *GeoLife* datasets, we further study to which extent the logical latencies can be reduced by reducing one parameter of our PLA compression scheme: the maximum segment length (set to 256 tuples in our evaluation).

Figure A19 shows the individual and combined average logical latencies as violin plots for different compressions measured over the (a) LiDAR (*Ford Campus*) and (b) Beijing GPS (*GeoLife*) datasets. The red violin plot on the

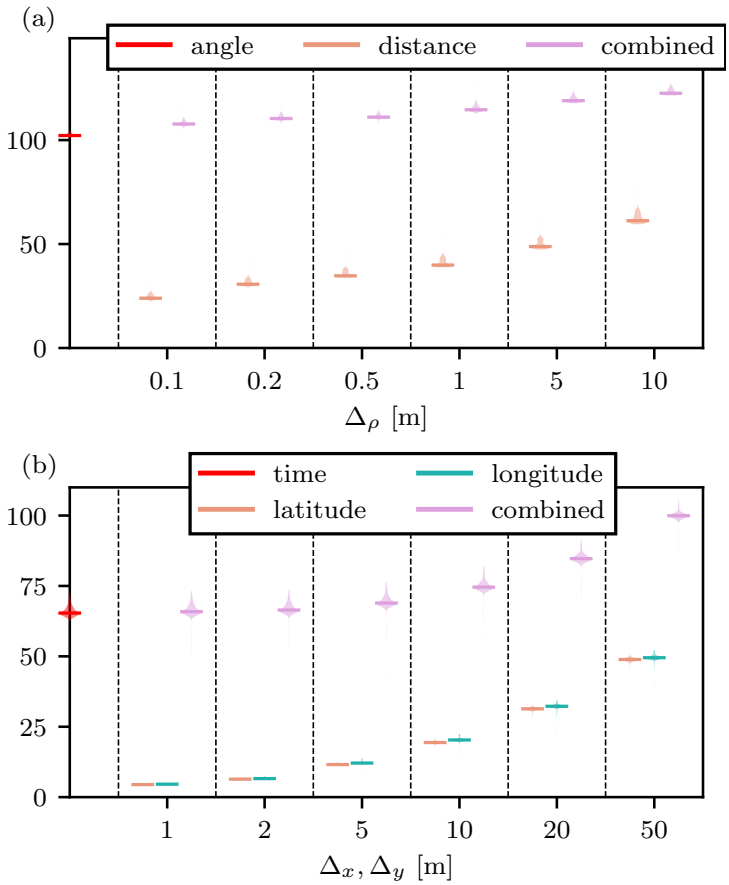


Figure A19: Distribution of logical latencies in number of tuples for (a) the LiDAR (*Ford Campus*) and (b) the Beijing GPS (*GeoLife*) dataset as a function of the respective maximum errors. The logical latency for the angle/time coordinate is displayed over the y-axis (red), as their corresponding maximum errors are constant over each of the two datasets.

left of both (a) and (b) displays the distribution of average logical latencies for the (logical) timestamps. As the (logical) timestamps are only compressed with a constant maximum error ($\Delta_\alpha = 0.0015$rad for LiDAR, $\Delta_t = 1$s for GPS), only one violin plot is shown per dataset for the (logical) time channel.

The span of the violin plots for different compressions is small for both (a) and (b), meaning that the logical latency depends more on the type of data than on the specific file of a certain datatype. Second, the combined logical latencies for both datasets are dominated by the latency of the timestamp channel. As this channel is quite linear, and thus easily compressible, we expect the longest segments for the timestamp channel and thus a large logical latency. Concretely, this means that other channels have to wait for the time channel to be decompressed before an original tuple can be reconstructed.

For the *GeoLife* GPS dataset, used in $Q_2 - Q_3$, the average difference between two timestamps in the original data is 5s. Thus, neglecting transmission time, it takes on the order of $100 \times 5\text{s} = 500\text{s}$ to reconstruct an original input tuple for GPS data with the aforementioned sampling rate.

For the *Ford Campus* LiDAR dataset, used in Q_1, there are 20000 readings of the logical timestamp channel α per second (see Section A4.1). Combined with a logical latency on the order of 100 tuples, this results in an average reconstruction time of at least 0.005s.

In the use cases investigated in this evaluation, those latencies carry little significance, as historic data is gathered. In these cases, where the data is replayed at much higher than live speeds, the transmission duration is dominant for the data gathering time. When live data is requested, however, the logical latency can lead to significant delays, but this is inevitable for PLA compression. The logical latency is strictly linked to the segment lengths of the PLA and can be reduced either via a smaller maximum segment length or via a smaller maximum error threshold, resulting in a PLA with shorter segments. For the combined logical latency, these changes will have greater effects if applied on the time channel, which due to its compressibility is dominant overall in our evaluation.

Figure A20 shows the logical latency and the compression for the LiDAR and GPS dataset (each averaged over all contained channels) for different values of the maximum segment length using constant error bounds (LiDAR: $\Delta_\rho = 1\text{m}$, $\Delta_\alpha = 0.0015\text{rad}$; GPS: $\Delta_x = \Delta_y = 10\text{m}$, $\Delta_t = 1\text{s}$). This figure motivates the choice of $n = 256$ for the maximum segment length, as for this value the compression is maximal. For higher n, the compression does not

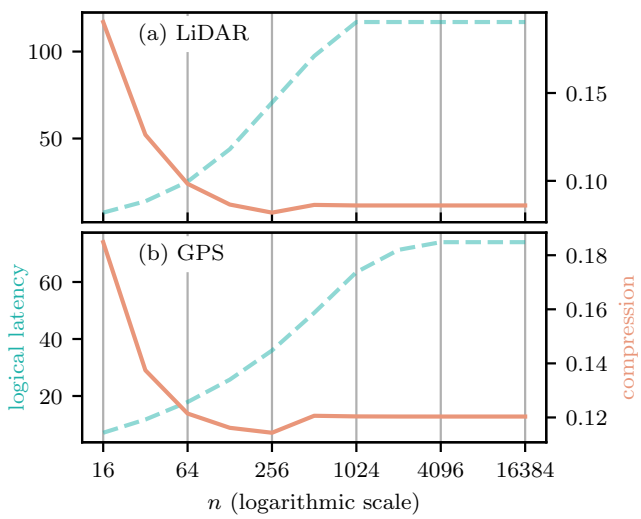


Figure A20: Average logical latency (over all channels) and compression for different maximum segment lengths n ($n = 256$ is the maximum segment length chosen in this evaluation).

increase further, as the maximal length of segments is an inherent characteristic of the data used (for a given maximum compressor error). The compression even becomes slightly worse as more data must be allocated for transmitting the segment length (i.e., two bytes are needed for $n > 256$). The logical latency increases with increasing segment length, and becomes stationary as the maximum inherent segment length is reached.
If lower logical latency is desired for a query, this figure shows that in turn lower compression will be achieved. However, depending on the region within the plots, large gains in logical latency can be achieved with comparatively smaller losses in compression, especially noticeable in the $n = 64 \dots 256$ region.

A5.7 Summary of Evaluation Results

The evaluation shows that, compared to the baseline, DRIVEN can maintain an adjusted rand index greater than or equal to 0.9 for the clustering of approximated data while compressing the raw data to less than 5%, 20% and 2.5% for LiDAR, GPS and a combination of GPS and other vehicular signals, respectively - outperforming lossless ZIP compression by factors of $2 - 10$ for LiDAR and GPS. Also, DRIVEN affords speed ups exceeding $\times 10$ in data gathering times for large-enough amounts of data (at least 200KB per sensor in our setup). Still, the logical latency inherent to PLA must be considered when working with live data. This logical latency can also be significantly decreased using an appropriate maximum segment length.

A6 Related Work

Clustering, as a core problem in data mining, has been extensively studied in the last decades (see e.g., the survey [102] and the references therein). The two main trends in clustering algorithms differ on what should be considered as a cluster, either privileging well-balanced ball-like clusters (as in the widely-studied k-means approach) or rather focusing on local density leading to arbitrarily shaped clusters (e.g., DBSCAN [66]-style). Other features that can distinguish existing clustering algorithms include their sensitivity to outliers (not interesting data that should be ignored in some applications), their ability to work with any distance function or the required level of parametrization.

Research on data streaming has also investigated how traditional batch-based clustering can be ported to the continuous domain. Clustering for large fast-coming streaming data has been widely studied in the last decade [177], focusing on producing approximations of the batch-clustering algorithm. Facing high-rate data streams, attention has indeed been paid to maintaining statistical summaries of the streamed data in order to generate on-demand clustering. Focus on recent data is captured by clustering only a *recent window* (using either landmarks, sliding windows, or assigning decreasing weights to older data) of points [177]. In [187], the authors design a fully streaming clustering algorithm (as the streaming version of a recently proposed clustering algorithm [168]), computing the exact same clustering of its batch-based counterpart. Similar to

the clustering algorithm described here, the clustering is density-based (hence for arbitrarily shaped clusters), works with any distance function but uses a different notion of dissimilarity between objects. However, contrary to our work, the ordering of data is not exploited resulting in $\mathcal{O}(n)$ time for the processing of a single point (while clustering n points).

Various solutions in the literature use approximation techniques together with streaming-based clustering methods to improve the performance, in one or more dimensions, of different clustering problems. Replacing time series by shorter representations [27] such as Discrete Wavelet / Fourier Transforms or Symbolic Aggregate Approximation to facilitate the processing and enhance the performance of several data mining algorithms (including clustering) has been a long trend in time series data mining [165]. Differently from our work, PLA or similar techniques (such as piecewise aggregate or piecewise constant approximation) are used to replace a time series by a lighter version to be later processed, as for clustering of time series in [128]. In our work, the objects being clustered are not the time series but the input points themselves and PLA is used to gather data efficiently (i.e., the data stream eventually clustered has the same length as the original one). To the best of our knowledge, this joint leveraging of streaming and PLA was not discussed before.

Concerning the generation of the PLA of a time series, there is an extensive literature covering it (e.g., [64], [111], [134], [201]) while focusing on different aspects of the approximation (errors, number of segments, processing time, etc.). Among recent works targeting sensor streams, we note the Embedded SWAB algorithm [22] (a modification of the well-known SWAB [111] segmentation) dedicated to the compression of wireless sensor raw data before transmission. The experimental study measuring power consumption shows that using PLA pays off in embedded devices by balancing out the computation overhead with reduced communication, thus reducing energy consumption. The authors note that the abstraction size is crucial in wireless sensor networks, thus motivating the study of trade-offs between small errors and high compression, which is one of the focal points of this work, in the context of the considered applications relevant in industrial settings. We also measure the time spent in the decompression process in our work and advocate that information retrieval from the measured data is also faster with PLA than with raw data transmission. In another recent work [81], the authors devise a PLA algorithm with a small memory footprint and average instruction count for resource-constrained wireless sensor nodes. They use a best-line approximation (similarly to us) but with no intercept (so more segments are produced), and the approximation error is bounded by segment instead of by point.

In an earlier work [90], a preliminary demonstration of DRIVEN can be found. In contrast to that work, we here propose a new algorithmic implementation of our multi-channels PLA compression that enables exact error guarantees through completely independent processing of time and other channels of a sensor and additionally resulting in better compression ratios. Moreover, we provide a more extensive evaluation that extends previous results to larger data volumes and higher network speeds. Furthermore, the present work investigates the effects and limitations of applying DRIVEN in live data

gathering scenarios and introduces an additional experiment advocating for the benefits of using PLA versus standard lossless ZIP compression.

A7 Conclusions

We have presented here the DRIVEN framework for data retrieval and clustering in vehicular networks. The framework, implemented in a state-of-the-art SPE, provides simultaneously an efficient way for gathering data and performing clustering on said data based on an analyst's queries. Information retrieval is achieved using PLA for compressing the input stream in a streaming fashion. Once uncompressed, the approximated stream is fed to a distance-based streaming clustering algorithm. Both the approximation and the clustering are parameterizable for allowing different applications to be run by the framework. Through thorough experimentation using real-world GPS and LiDAR data as well as other vehicular signals, we show the versatility of the framework in being able to answer different types of queries of historical data involving various clustering requests for vehicular networks, and also show that compression in data retrieval speeds up the transmission of gathered data while being able to preserve a very similar clustering quality compared to raw data transmission. Data can be reduced to $5-35\%$ of its raw size, reducing drastically the duration of the gathering phase for large volumes of data, with only a small accuracy loss on the clustering.
We furthermore have evaluated the application of DRIVEN in a live-data scenario and studied the additional (and inherent) latency from the PLA compression, which can nevertheless be reduced for a predictable loss in compression, and gauged the compression capabilities of our PLA implementation using ZIP compression.

The idea behind DRIVEN is to leverage the cumulative power of edge devices to improve data analysis applications that are traditionally deployed entirely at data centers and that require all input raw data to be first gathered centrally. The solution we propose in this paper can be enhanced along different dimensions in future work. First, other techniques (e.g., Symbolic Aggregate Approximation - SAX) can be leveraged at the vehicles to reduce the amount of data to be forwarded, and it is thus interesting to study how such techniques would perform along with the performance metrics we take into account in this paper. Second, given that many other machine learning techniques are commonly used in cyber-physical systems' data analysis, their integration (and possible porting to the streaming paradigm) within DRIVEN is also of interest. Finally, it is also important to notice that the computational power of each edge device (be it a vehicle or something else) can be used in conjunction with the data center's one in several ways. While we study a solution that leverages the edge device's computational power to approximate and compress raw data, we also believe that distribution of machine learning tasks (e.g., learning over different subsets) is a way to leverage such computational power that is worth exploring and can enable efficient and effective solutions.

Chapter B

Time- and Computation-Efficient Data Localization at the Edge

Romaric Duvignau, **Bastian Havers**, Vincenzo Gulisano, Marina Papatriantafilou

The following is an adapted version of the work published in *IEEE Access, Vol. 9, p. 137714-137732*, as *"Time- and Computation-Efficient Data Localization at Vehicular Networks' Edge"*. Any changes serve only to retain the consistency of this thesis.

Abstract

As Vehicular Networks rely increasingly on sensed data to enhance functionality and safety, efficient and distributed data analysis is needed to effectively leverage new technologies in real-world applications. Considering the tens of GBs per hour sensed by modern connected vehicles, traditional analysis, based on global data accumulation, can rapidly exhaust the capacity of the underlying network, becoming increasingly costly, slow, or even infeasible. Employing the edge processing paradigm, which aims at alleviating this drawback by leveraging vehicles' computational power, we are the first to study how to localize, efficiently and distributively, relevant data in a vehicular fleet for analysis applications. This is achieved by appropriate methods to spread requests across the fleet, while efficiently balancing the time needed to identify relevant vehicles, and the computational overhead induced on the Vehicular Network. We evaluate our techniques using two large sets of real-world data in a realistic environment where vehicles join or leave the fleet during the distributed data localization process. As we show, our algorithms are both efficient and configurable, outperforming the baseline algorithms by up to a 40× speedup while reducing computational overhead by up to 3×, while providing good estimates for the fraction of vehicles with relevant data and fairly spreading the workload over the fleet. All code as well as detailed instructions are available at `https://github.com/dcs-chalmers/dataloc_vn`.

B1 Introduction

With the recent advancements in connected Vehicular Networks [44], often facilitated by Vehicular Ad Hoc Networks (or VANETs) [107], the automotive industry is witnessing an unprecedented growth of possible ways for leveraging the fine-grained data sensed in modern vehicles and enhance drivers' safety and experience. If accessed in a real-time fashion as it is being sensed, such data can lead to fresh, up-to-date insights for analysts and practitioners [153]. Similarly to how Mobile Edge Computing [196] pushes parts of data analysis applications, previously run entirely in the cloud, towards mobile users, to achieve lower latency and bandwidth consumption, Vehicular Edge Computing [130] aims at better utilizing the cumulative computational power of Vehicular Networks while coping with high-mobility networks and the challenges stemming from their dynamic topologies and communications. When focusing on data analysis in the context of Vehicular Networks, a critical challenge is that of data gathering [91], [119] for subsequent analysis. While in the past companies could potentially afford the central gathering of all the data sensed by an entire fleet of vehicles (see [73], [175] for applications and services in VANETs), a modern vehicle can now generate several gigabytes of data per hour [44], making this approach infeasible in terms of infrastructure and costs - which could be alleviated by enforcing the collection of just enough data from relevant vehicles only.

Several recent studies in the literature are focusing on how to avoid general central data gathering by transitioning to models in which the data selection processes, or even the analysis itself, are pushed towards the vehicles [46], [152], [188], for efficient and continuous filtering [150], preprocessing through online compression [58], [91], and conversion of raw data into information in Federated Learning [139]. However, these distributed analysis models often lack mechanisms that attempt to involve only valid vehicles into the analysis, to thus avoid unnecessary computational load and data transfers or other hindrances to the analysis. As an example, Federated Learning on vehicles [132] requires the involvement of vehicles that have gathered sufficient suitable data in order not to hinder the learning process [212]. Furthermore exacerbating the issue of vehicle selection are skewed data distributions on vehicles [53], [63] and data minimization directives such as the European GDPR, which dictate to minimize exposure risk and thus overall involvement of customer vehicles. To find vehicles possessing data relevant to an analysis task, one has to overcome the lack of a-priori knowledge about which vehicle has collected which data, without centrally gathering said data first, by leveraging the vehicles' computational power. As vehicles possess only application-specific computational hardware that is not provisioned for more general tasks, it is furthermore paramount to avoid unnecessary computational strain on the fleet.

Contribution

In light of the present challenges concerning the transition from central to distributed, edge data gathering and analysis in large fleets of vehicles, we pose the following question:

How can data residing on the edge nodes of a Vehicular Network be localized efficiently through request-spreading from a central coordinator to the vehicles?

The manner of spreading requests is regulated by a data localization algorithm orchestrated at a central coordinator that has to be aware of the completion time and the computational overheads induced on the fleet of vehicles. Once a request is sent from the coordinator to a vehicle, the latter checks locally whether a set of conditions is satisfied by the stored data (e.g., whether the data spans a given time interval, or whether the data indicates that the vehicle is associated with a specified geographical position, speed, driving mode, etc.), and returns a compact answer indicating whether the conditions hold. When performing traffic flow analysis, for instance, this could be used to efficiently compute a certain statistic (e.g., the average speed) only based on vehicles driving above a certain speed, within a city center, or during rush hour, or to mark these vehicles for a subsequent analysis. With the ultimate goal of collecting a certain amount of answers from vehicles matching a given set of conditions, we propose efficient *data localization algorithms*, that can also cope with dynamic connectivity, and benchmark them against baseline algorithms. Our evaluation, based on realistic queries and also assessing the spreading of requests using real vehicular data, shows that our data localization algorithms provide up to a 40× speedup and less than one-third of the computational overhead, compared with baseline algorithms optimizing only one of the metrics.

The typical characteristics of Vehicular Networks include recurrent topology changes due to vehicles' high-speed mobility and properties of the underlying road network (including communication-challenging environments such as bridges, tunnels, etc.). Though this challenging aspect can impede successful communication between a central coordinator and vehicles, our work accounts for it with algorithms able to react rapidly to dynamic connectivity issues. To the best of our knowledge, we are the first to formulate and analyse the problem of localizing data in a vehicular fleet, as well as to propose algorithms that can tune the key trade-off between resolution time and overhead on the vehicles and the communication network. The remainder of the paper is organized as follows: We introduce the System Model in Section B2. We present in Section B3 baselines and propose novel algorithms for solving data localization queries by spreading a set of requests over a fleet of vehicles. We lay out our evaluation methodology in Section B4.1 and cover the evaluation of the proposed algorithms in Section B4.2. We discuss related work in Section B5, before concluding the paper with a summary in Section B6. In the Appendix, we discuss the relation of the present paper to an earlier conference article that presented first results on a preliminary formulation of the problem for a subset of the system types considered here.

B2 System Model and Problem Statement

B2.1 Problem Definition

We consider the following model: a *fleet* V of k vehicles, also referred to as *nodes*, is equipped with different types of sensors $s_1, s_2, \ldots$ from the fixed sensor set S, e.g., $S = \{\text{GPS, steer, break, } \ldots\}$. We define the continuous timestamped record sequence recorded by the sensor $s_i \in S$ at vehicle $v \in V$ as

$$s_i(v) = (t_0, x_0), (t_1, x_1), \ldots$$

where x_j is the sensor reading at time point t_j. All vehicles are connected via a two-way communication channel to a central *coordinator* C, e.g., a datacenter. Data analysts require C to process *data localization queries* $q_1, \ldots, q_r$, with each query focusing on some subset of the possible sensors for some time span of recorded data.

A *(data localization) query* here corresponds to the task of identifying n vehicles in the fleet with relevant data. In more detail, a query q carries a specific *condition* P that must be fulfilled by vehicles' data to be *relevant* for q and every query specifies some number of positive *answers* n (responses from distinct vehicles with relevant data, where a "positive" or yes-answer implies that P holds locally) that must be collected to *resolve* q before a potential next analysis step involving only these n vehicles with relevant data can follow. k_q is the number of vehicles in the fleet on which P holds, and $QR = k_q/k$ is the query rate or *average answer rate* of a query q. We assume $k \gg k_q > n$, thus n vehicles with relevant data can indeed be found for a query q. To check if a particular vehicle v fulfills q's condition P for some query q, a *request* $r(q)$ is sent to v and after checking P locally, v responds to C with its yes- or no-answer (and potentially some additional data). If a vehicle v receives a new request $r(q')$ corresponding to another query q' while already processing a previously received one $r(q)$, then $r(q')$ is added to v's local *task queue*; once v has terminated its processing of $r(q)$, v's task queue is then processed in FIFO order. Naturally, not every vehicle can answer positively to every request, because of lack of data or because the data is found not suitable to answer that particular query. The required number of answers is meant to localize a sufficient amount of data from the vehicles to be meaningful for the analysis task at hand, while avoiding excessive participation.

Notice that contacting exactly the number of vehicles given by the analyst is not necessarily enough, as some might not answer or have data that does not satisfy the condition. On the other hand, contacting too many vehicles might result in some of them wasting some of their computational power to inspect data that is not actually needed by the analyst. Thus, we need to require just enough positive answers (e.g., for statistical significance or for reducing the likelihood of identifying individuals in the data), but not too many (because of the time needed to collect all the data [90], [91], the induced computational load, and potential network stress).

We will use the notation $q.P$ for the condition and $q.n$ for the minimum number of answers required to complete the data localization query q. The

condition P specifies (i) which sensors are relevant to the query and (ii) an overall condition that local data must satisfy for the vehicle to acknowledge that its data can be part of the desired analysis. In the following: $S_q \subseteq S$ is a subset of sensors which are relevant for the query q (by default $S_q = S$); and (t_{start}, t_{end}) are time bounds spanning q's time interval of interest such that for every participating node v and every sensor of type $s_i \in S_q$, only the portion of local data $\{(t, x) \in s_i(v) \mid t_{start} \leq t \leq t_{end}\}$ is examined (if not specified, the full recorded data is considered).

Example query To interactively check the traffic flow within a certain area A of a city, an analyst wishes to identify $n = 100$ vehicles in a $k = 100000$ vehicles fleet, with $P =$ "driven within area A in the last hour, with an average speed greater than 50km/h over a 10 minutes window, and with GPS measurements spaced by at most 5s from each other". In this query, assuming the query is created at 9:00, $S_q = \{\text{GPS}\}$, $(t_{start}, t_{end}) = (8{:}00, 9{:}00)$, and A is a bounding box or geofence approximating the area of interest. Notice that, in this example, to check if any period of 10 minutes (representing only 12000 data points for 100 cars with 5 seconds GPS readings) within the last hour fulfills the condition, a centralized solution requires between $n \cdot 3600/5$ (considering at least n vehicles need to answer) and $k \cdot 3600/5$ data points to be transmitted, i.e., 72000 to 7200000 data points; that is, in order to check the existence of any period of 10 minutes of consecutive readings with speed greater than 50km/h, since the coordinator has no information whether they exist (and in which portion of the hour), the entire hour needs to be retrieved and checked. Checking the condition on-board the vehicles alleviates this data transfer and only short yes/no answer messages need to be communicated with the coordinator.

Applications In the aforementioned example query, the mere collection of affirmative or negative answers from the fleet can already provide a good estimation of the fraction of vehicles that satisfy P in the fleet (and thus quantify the traffic flow in the area in question). This defines a first set of applications, in which a query itself gives rise to a statistical insight by providing a population estimate. As hinted in the example, transmitting raw data from a random sample of the fleet in the above example to check at the coordinator whether P holds for a vehicle would incur significantly higher communication costs, while yielding the same insight. In addition to the first set of applications, the coordinator node C can ask the vehicles which answer positively to subsequently perform tasks suitable only to them. These tasks can include transmitting raw data, performing statistical summaries such as averages or other aggregate functions over the local data, or higher-level computations on-board the vehicle over the relevant data, such as training an Artificial Neural Network. However, we do not consider the query post-treatment in this work and concentrate on the aspect of data localization. That is, we focus on finding a suitable set of vehicles that will participate in the query resolution process. One may note that some of the subsequent tasks could be answered alongside P's verification, entailing minor changes in the processing time of the query. The aggregated value could be transmitted with the vehicle's yes/no-answer to C without significantly

changing the resolution time. For computationally heavier analysis tasks or those requiring substantially longer time (such as data transfers from the vehicles), we consider that the post-treatment is executed in a way independent of the selection process, i.e., vehicles will always first inform C if they validate P before executing the post-treatment.

B2.2 Fleet Model

The fleet V of vehicles that can be contacted encompasses the totality of vehicles that are equipped to take part in answering requests incoming from the coordinator C. As vehicles in V may be switched off, we introduce the *active* set of vehicles $V_t \subseteq V$ at a time t as the ones switched on and willing to participate in the data localization queries' resolution process. A realistic fleet is a dynamic entity where vehicles leave and join impromptu (thus, vehicles are being switched on and off). We differentiate two variants of the underlying system model depending on how the fleet V_t evolves during the resolution of a batch of queries.

We first consider a *static fleet* model. In that model, the set of contactable participants is always fixed, hence we do not consider that new vehicles may join the fleet or that vehicles may leave within the time interval spent resolving a particular query. Thus, $V_t = V$ for all t.

In the *dynamic fleet* model, the set V of *all* vehicles (such as all vehicles from a company fleet) is larger than the active set V_t at time t, unlike in the static model. The active set evolves through time with the possibility for new vehicles to join and for old ones to leave. Contrary to the static model, a vehicle may be switched off (that is, leaving the active set) during the resolution of a particular query q and may not send its answer for q to C; similarly, new vehicles that were absent at the start of q's resolution may join the active fleet at any time during q's resolution process.

To quantify the amount of vehicles leaving the fleet over time, we rely on the notion of *churn*. Given a period Δ, we define the churn at time t based on the number of vehicles that leave the fleet during the period $[t - \Delta, t)$ but that were part of the fleet during the period $[t - 2\Delta, t - \Delta)$. More formally:

Definition B1. $\text{Churn}_\Delta(t)$: *The number of vehicles that were part of $V_{t'}$ for all $t' \in [t - 2\Delta, t - \Delta)$ but that leave the fleet at any $t'' \in [t - \Delta, t)$, divided by the size of V_t.*

Note that while the churn takes into account only vehicles leaving the fleet, the number of vehicles joining the fleet can be obtained through the size of the active fleet and the value of the churn.

B2.3 Communication Model

We assume that the coordinator C has no access to the vehicles' local data other than through communication with them; thus, the amount of work needed to test $q.P$ for a query q cannot be estimated before checking P locally on the appropriate vehicle. We further assume that C will always successfully contact

any active vehicles and as soon as a vehicle does not communicate for a certain period of time, it is considered inactive.

In the likely event that the local requested data is missing, the involved vehicle does not satisfy $q.P$ and it answers negatively. Similarly, active vehicles unwilling to participate in a query's task (for privacy or other reasons) can be modeled by negative answers.

In the dynamic fleet model, new vehicles signal their presence once they become active and ready to answer potential requests. Since connection to C can be lost at any point in time (e.g., driving through a tunnel or reaching a poorly covered geographical area), we assume that once a vehicle v has become inactive, it cannot be reached by C and drops all currently processed queries; hence, C will not receive any answers from v for the queries it was processing at the time. This allows abstracting the high degree of node mobility and its effect on communication by possibly short interruptions in the active status of the vehicles; here, individual packet losses are neglected as overall they can be compensated by configuring a lower transfer rate and higher latency for the underlying communication channel.

B2.4 Performance Metrics

We associate with each query three performance metrics:

1. **Query Resolution Time**, the elapsed time between deploying a particular query q at C and q's resolution, i.e., when $q.n$ positive answers have been collected at C.

2. **Fleet Workload**, the overall computing load on the vehicles defined as the sum of individual processing times (local workloads) of all vehicles involved in processing the received requests associated with the query.

3. **Fairness** of the algorithms, the standard deviation of the cumulative local workloads between the vehicles that received a request, representing how fair the spread of the fleet workload is.

Notice that optimizing for both (1) and (2) at the same time is not straightforward, as they compete. More concretely, query resolution time is minimized by simply asking all vehicles in the fleet and ignoring answers after n positive answers are retrieved (thereby maximizing the Fleet Workload required per query, which is further exacerbated when queries are executed in parallel or are computationally expensive, see Section B4.1.3) while the fleet workload is minimized for instance by asking one vehicle at a time in a round-robin fashion (implying high time overhead).

The amount of uncertainty in the model is an additional challenge: each query requires different amounts of time per vehicle that can hardly be foreseen. The reasons for this are twofold:
(i) *Query semantics.* As it is unknown how much relevant data a vehicle has collected, it can not be estimated beforehand how long it will take for this vehicle to search for the property required by the request. Likewise, some queries may be answered positively as soon as the first matching instance is

found in the data, whereas a negative answer requires checking all potentially relevant data.
(ii) *Computing capacity.* A vehicle that is contacted may be performing other processing with higher priority and equally unknown completion time before it can start answering the latest received request at hand.

B3 Data Localization Algorithms

We present here algorithms that select a subset of vehicles among the k_q vehicles satisfying P for a *single* query $q = (P, n)$ assuming $k_q \geq n$. For a set of queries $q_1, \ldots, q_r$, each query can be resolved by executing at C, either sequentially or concurrently, the procedures described hereafter. Without further assumptions on the distribution of nodes satisfying P, it is natural to randomly and uniformly send requests to nodes within the pool of nodes that have not been requested yet. However, other factors (such as the number of requests currently being processed on the vehicle, historical local computation load, etc.) can be used to bias the selection process. Since in our setting, queries are relatively short to solve (from a few seconds to minutes at most), we consider that algorithms do not need to send another request to a vehicle that has answered negatively, in case its newest acquired data now satisfies the property P. We hence assume in our analysis that a vehicle's to a query q does not change over the whole period of the algorithms' execution.

We present in this section four algorithms focusing on different measures:

- BaseEager, a baseline approach that optimizes the resolution time needed to answer q,
- BaseLazy, a baseline approach that optimizes the number of contacted vehicles (hence minimizing required communication and reducing fleet workload), and ensures no more than n positive answers are ever received,
- BalanceRequests, a new approach that balances the trade-off identified through the two baseline algorithms, in order to quickly collect n answers without inducing excessive load on the vehicular nodes, and
- BalanceLoad, an approach that extends BalanceRequests by prioritizing the least-used vehicles during the selection process to balance the workload.

The Base* algorithms introduced in this work are meant to benchmark the Balance* algorithms against edge cases (i.e., optimizing only one aspect) of the spectrum of possible trade-offs.

The four algorithms can maintain the following sets in each execution:

- $F \subseteq V$, the set of contacted vehicles since the beginning of the algorithm;
- A, the set of all answers from vehicles $v \in F$ that have been received by the coordinator;

- $R \subseteq A$, the subset of positive answers (where each answer contains whether P holds plus metadata identifying the sending vehicle, see Section B2.1) among all the ones received.

We begin by introducing these algorithms in the context of an idealized model in which the fleet is static and the on-board execution is synchronous, before presenting the algorithms in our complete static (Section B3.2) and dynamic model (Section B3.3).

B3.1 Data Localization in the Synchronous Static Model

To ease the introduction of the algorithms, we consider in this subsection a synchronous model (in the next subsections we present the generalization of the algorithms for the asynchronous model): communications with C are instantaneous and all nodes need a constant amount of time to check the property P, i.e., one "round" is the time to check any request on one vehicle. Hence, after a round of time has elapsed, C has received answers (yes/no) from all nodes that were asked during that round. In this simplified situation, only two aspects have to be considered in order to measure the performance of data localization procedures: (1) the total number of rounds needed at C to receive n answers, and (2) the number of nodes m that have checked if $q.P$ holds (which is equivalent to the fleet workload on the vehicles, since each contacted vehicle has spent exactly one round checking the request). Note that for clarity we discuss the behavior of the algorithms in the case of answering a single data localization query q.

BaseEager - synchronous, static

Presented in pseudocode in Algorithm B1, BaseEager aims to optimize a query's resolution time, by immediately querying all available nodes; this resolves the query in a single round (under our assumption that enough vehicles with relevant data are in the fleet). Indeed, consider any other algorithm $\mathcal{A}$ that does not contact at least one node v during the first round. In the situation that $k_q = n$ and P holds on v, only $k_q - 1$ answers are received after a single round of communication and a second one is required to retrieve all required

Algorithm B1 BaseEager

```
1: function BaseEager(V, q)                    ▷ fleet V, query q with n = q.n
2:     R ← ∅                                   ▷ set of collected positive answers
3:     for v ∈ V do
4:         send(q, v)                          ▷ send request r(q) to vehicle v
5:     while |R| < n do
6:         r ← receive()                       ▷ block till receiving next answer
7:         if positive(r) then
8:             R ← R ∪ {r}
9:     output R
```

answers; hence, $\mathcal{A}$ is not optimal in regards of the resolution time. Executing Algorithm B1 to obtain all needed answers leads nonetheless to a large strain on the vehicular nodes. In particular, the number of queried nodes is always k, independently of k_q and n. Thus, all nodes always participate in q's resolution, even though the number of required answers n might be relatively small.

BaseLazy - synchronous, static

This Algorithm is presented in pseudocode in Algorithm B2. The focus of this algorithm is on reducing the computational overhead and communication induced on the fleet. To ensure that only the minimum number of nodes are being requested to check P, one must ask at most as many new nodes as the number of currently missing positive answers. Any algorithm satisfying such an assertion is associated with a minimal fleet workload, and the best algorithm in this category selects randomly as many nodes as possible by asking m "new nodes" for each round where there are m missing answers. Once n positive answers are received, the procedure stops. In an edge case, every vehicle in the fleet has to be contacted to achieve this. Since the algorithm contacts m vehicles per round, the algorithm will proceed the slowest for $m = 1$. When $(n-1)$ yes-answers are collected in the first round and the last yes-answer is obtained after asking every single other vehicle in the subsequent rounds, this results in a resolution time of $k - n + 1$ rounds.

However, on average BaseLazy requires fewer rounds, as claimed by Proposition B1 below. When sending a request to a vehicle that has not participated so far, the probability of receiving a yes-answer is in general $(k_q - r)/(k - f)$ where $f = |F|$ is the number of vehicles already requested and $r = |R| \leq n$ the number of positive answers already received. In the following, we assume for simplicity that the probability of obtaining a yes-answer is constant and equal

Algorithm B2 BaseLazy

```
1:  function requestRandVehicle(q, F)
2:      v ← random(V, F)                    ▷ random vehicle in V excluding F
3:      send(q, v)
4:      output v

5:  function BaseLazy(V, q)                 ▷ fleet V, query q with n = q.n
6:      F ← ∅                               ▷ set of asked vehicles
7:      R ← ∅                               ▷ set of collected positive answers
8:      for 1 ≤ i ≤ n do
9:          F ← F ∪ { requestRandVehicle(q, F) }
10:     while |R| < n do
11:         r ← receive()
12:         if positive(r) then
13:             R ← R ∪ {r}
14:         else
15:             F ← F ∪ { requestRandVehicle(q, F) }
16:     output R
```

to $QR = k_q/k$ during the full execution of the algorithm; this corresponds to our typical use-case where n is much smaller in comparison to both k_q and k. The number of rounds used by the algorithm can then intuitively be shown to be logarithmic by the following reasoning: When requesting x vehicles in a round, approximately $x \cdot QR$ positive answers are received, thus if x_i denotes the number of requests sent in round i, we have $x_i \approx x_{i-1} \cdot (1 - QR)$, as vehicles answering negatively trigger another request. Setting $x_1 = n$, we get $x_i \approx n \cdot (1 - QR)^i$ and we obtain $r \approx \log_{1/(1-QR)}(n)$ when requiring $x_r = 1$ (when the algorithm roughly concludes). More formally, we argue that:

Proposition B1. *BaseLazy solves a single query in the static, synchronous case on average in $\mathcal{O}(\log(n))$ rounds.*

Proof. Let us visualize the query resolution as follows. Let $\mathcal{P} = p_1, \ldots, p_n$ be n random processes that aim to retrieve one answer each to the query, and each process will remain active until it acquires a positive answer. BaseLazy can be seen as sending one request per round for each process that is still active and doing nothing for the ones that have already got a yes-answer. Under our assumptions, during a certain round, $p_i \in P$ retrieves a positive answer with constant probability $p = QR$ and a negative answer with probability $1 - p$. Each process, being independent of the others, will need $1/p$ rounds on average to acquire a yes-answer (geometric distribution with parameter p). The number of rounds $M(p, n)$ that is necessary for all processes to stop is thus the maximum of n independent *geometric random variables* of parameter p. The expected value $\mathbb{E}\left(M(p, n)\right)$ is known [62] to be bounded by

$$\frac{H_n}{\ln \frac{1}{1-p}} \leq \mathbb{E}(M(p, n)) < \frac{H_n}{\ln \frac{1}{1-p}} + 1$$

where H_n is the n-th harmonic number. Using $H_n = \ln n + \mathcal{O}(1)$, one obtains that $\mathbb{E}(M(p, n)) = \log_{\frac{1}{1-QR}}(n) + \mathcal{O}(1)$ for a constant QR.

□

Similarly, BaseLazy sends on average requests to significantly fewer vehicles than in the extreme case, as shown by the following Proposition B2:

Proposition B2. *BaseLazy solves a single query in the static, synchronous case asking on average n/QR vehicles.*

Proof. Following the presentation of the previous proof, the number of requested vehicles is obtained as the sum of n independent geometric random variables, each of which has an expected value of $1/p = 1/QR$. By linearity of expectation, we obtain that n/QR vehicles will receive a request. □

BalanceRequests - synchronous, static

We introduce now an efficient scheme to achieve a low fleet workload while resolving queries within few processing rounds, balancing the tradeoffs of BaseEager (high workload) and BaseLazy (slow query resolution time). The

Algorithm B3 BALANCE*-skeleton

```
1:  function BALANCE*(V, q, α, β)                ▷ fleet V, query q with
                                                   n = q.n, α > 0, β ∈ (0, 1]
2:      p ← 1                                    ▷ estimation of probability to answer yes
3:      F ← ∅                                    ▷ set of requested vehicles
4:      A ← ∅                                    ▷ set of collected answers
5:      R ← ∅                                    ▷ set of collected yes-answers
6:      while |R| < n do                         ▷ until n answers are collected
7:          askNewBatch(α, p)
8:          while waitForAnswers() do
9:              receiveAndUpdate()
10:         p ← max{ |R|/|A|, 1/(|A|+1) }        ▷ update probability
11:     output R
```

main idea behind BALANCEREQUESTS is to employ QR, the share of vehicles in the fleet on which $q.P$ holds, to scale the number of vehicles contacted in each round such that the expected number of positive answers is equal to the number of total outstanding positive answers. As QR is unknown during the execution of the query, we replace it with the running estimate $p = |R|/|A|$, and show in Section B4.2.4 that p gives a reasonable estimation of QR.

We will present various implementations of the BALANCE* algorithms based on the skeleton algorithm shown in Algorithm B3, which proceeds as follows: Keeping track of p, F, A, R, the algorithm concludes by returning R, the set of yes-answers. To achieve this, the algorithm begins by contacting a new batch of vehicles in ASKNEWBATCH(), and then proceeds to receive answers from the contacted vehicles using RECEIVEANDUPDATE until WAITFORANSWERS() evaluates to $false$. At this point, the algorithm updates the value of p using the answers received so far, and loops back to the beginning; the loop is continued until $|R| = n$, i.e., a sufficient number of yes-answers has been acquired.

In the synchronous model and with a static fleet, algorithm BALANCEREQUESTS employs those variants of ASKNEWBATCH() and WAITFORANSWERS() described in Algorithm B4. ASKNEWBATCH() has as input the running estimate p and a parameter α, and proceeds as follows: m marks the currently missing yes-answers $n - |R|$. We denote by

$$\ell = \left\lceil \alpha \cdot \frac{m}{p} \right\rceil \tag{2.1}$$

the adjusted expected number of vehicles to contact to receive the outstanding m answers, employing the running estimate p. The parameter $\alpha > 0$ allows the algorithm to depart from the estimated expected number of vehicles to contact to get the remaining answers, by sampling more or fewer vehicles. This allows to either shorten (when $\alpha > 1$) the average number of rounds needed to resolve q while potentially increasing the fleet workload, or on the contrary (when $\alpha < 1$) to slow down q's processing by being more prudent and avoiding requesting more vehicles than necessary (and thus getting closer to receiving exactly n answers at the end). Having contacted ℓ vehicles or exhausted the fleet of vehicles, the function returns. BALANCEREQUESTS then enters a loop of

Algorithm B4 BALANCEREQUESTS - synchronous, static

```
1: procedure askNewBatchAux(α, p, m)
2:     ℓ ← ⌈α · m/p⌉                                  ▷ new vehicles to contact
3:     for 1 ≤ i ≤ ℓ do
4:         if |F| < k then                             ▷ if fleet not exhausted yet
5:             F ← F ∪ { requestRandVehicle(q, F) }

6: procedure askNewBatch(α,p)
7:     m ← n − |R|                                    ▷ remaining yes-answers to collect
8:     askNewBatchAux(α, p, m)

9: procedure receiveAndUpdate()
10:    r ← receive()
11:    A ← A ∪ {r}
12:    if positive(r) then
13:        R ← R ∪ {r}

14: function waitForAnswers()
15:    output |A| ≠ |F|
```

receiving answers until waitForAnswers() returns *true*, i.e., until all contacted vehicles have sent an answer.

Following the general logic of the BALANCE* algorithms described in Algorithm B3, the value of p is then updated as

$$p = \max\left\{\frac{|R|}{|A|}, \frac{1}{|A|+1}\right\}, \tag{2.2}$$

where the second case is used when no positive answers have been received during the first round(s), i.e., $|R| = 0$. A next batch of vehicles is then contacted, until n yes-answers are received.

BALANCELOAD - synchronous, static

This algorithm does not have an equivalent in the synchronous model, as it attempts to balance the individual workloads of each vehicle. In the synchronous model, the workload of each vehicle answering a request is identical by assumption (as all vehicles answer a request synchronously). We thus defer introducing this algorithm to the following section.

B3.2 Data Localization in the Asynchronous Static Model

We now generalize the algorithms presented in the previous section to the asynchronous data localization model, i.e., when request processing time is both vehicle- and context-dependent. In a typical vehicular environment, one cannot generally assume bounds on neither the time a vehicle needs to process a request nor on the communication delays in the network. Consequently, the algorithms have to adapt to the following scenarios: (1) how to avoid being

blocked by the slowest-answering vehicles; and (2) how to deal with vehicles that answer late? While BaseEager and BaseLazy achieve their respective goals without adaptations in the asynchronous model, we tune BalanceRequests to the asynchronicity and furthermore extend it to yield BalanceLoad. In addition to asynchronicity, we now also extend to the more general case of more than a single data localization query deployed simultaneously. As a reminder, vehicles possess a FIFO task queue (see Section B2.1) in which incoming requests are stored and processed sequentially.

BaseEager - asynchronous, static

Shown in pseudocode in Algorithm B1, BaseEager optimizes the time required to answer a single query q by contacting all vehicles upon receiving it. In our asynchronous model, the query resolution time then needed for a single q is the best possible and corresponds to the n-th fastest positive answer received at C. In contrast to that, the fleet workload is also the highest possible, as all k vehicles have processed $r(q)$. Note that the guarantee on fastest resolution does not hold in the case of multiple simultaneous queries: Let us assume that vehicle v gives the n-th fastest yes-answer to query q in the single-query case. However, when $r(q)$ is received by v, v is busy processing another request $r(q')$; thus v will wait before answering $r(q)$, which would conclude the query q. Thus, the execution time of q is dependent on the presence and order of other concurrent queries on the requested vehicles.

BaseLazy - asynchronous, static

Shown in pseudocode in Algorithm B2, BaseLazy optimizes the number of requested vehicles (hence minimizing needed communication to spread all requests) by contacting a new vehicle only when strictly needed. Since in our asynchronous model it is not guaranteed nor assumed that vehicles will have similar answer times (only that they will answer at some point), this algorithm does not necessarily imply a minimum load on the network. Indeed, it might be the case that requesting more vehicles that require shorter processing times to answer will use fewer resources overall.

Algorithm B5 BalanceRequests - asynchronous, static

```
1: procedure askNewBatch(α,p)
2:     m ← n − |R| − p · (|F| − |A|)        ▷ remaining yes-answers, corrected by late ones
3:     askNewBatchAux(α, p, m)              ▷ cf. Algorithm B4
4: procedure receiveAndUpdate()             ▷ same as Algorithm B4
5: function waitForAnswers(β)
6:     output |A| < β · |F|                 ▷ share β of contacted vehicles has answered
```

BalanceRequests - asynchronous, static

The asynchronous variant of BalanceRequests (Algorithm B5) is similar in essence to its round-based version described in Section B3.1 but needs to take into account that not all contacted nodes reply at the same time (as they do in the synchronous model). To do so, we change the behavior of the function waitForAnswers(), as shown in Algorithm B5; it accepts a new parameter $\beta \in (0, 1]$: a certain proportion of answers over all requested vehicles that we will wait to receive before re-evaluating the running estimate of yes-answer share p and proceeding to the next batch of selection (see Algorithm B3). When $\beta = 1$, the algorithm waits for the reception of all answers before continuing; this is effectively the case in the synchronous model, Section B3.1. Setting a lower value for β allows us to make a decision without having to wait for the slowest vehicles. Another change is about taking into account vehicles that have not yet answered when a new iteration starts. Based on previously received answers, we estimate that a fraction p of the $|F| - |A|$ requested vehicles that have not yet answered, will eventually answer positively while the next batch of vehicles is already being sent requests. This allows to reduce the number of vehicles asked in the next iteration, and thus reduce excessive participation. This change is applied in askNewBatch() (Algorithm B4): we adjust the number of vehicles to contact next, ℓ, by

$$\ell = \left\lceil \alpha \cdot \left(\frac{n - |R|}{p} - (|F| - |A|) \right) \right\rceil \tag{2.3}$$

Those $\lceil p \cdot (|F| - |A|) \rceil$ vehicles are hence counted as *expected* answers when calculating ℓ.

BalanceLoad - asynchronous, static

This is a variation of the previous algorithm that presents a further refinement of vehicle selection, differing in how the ℓ vehicles are selected during each batch in askNewBatch(), as shown in Algorithm B6. Instead of randomly selecting new nodes to request, vehicles having low local workload or involved in only a few concurrent data localization queries are picked first in the selection phase. The main difference with Algorithm B5 is that instead of requesting a random vehicle using requestRandVehicle() among the not yet requested ones (see requestRandVehicle() in Algorithm B2), vehicles are selected in the order of their lowest *local workload* measured as (1) number of simultaneous requests being processed on the vehicle (for the concurrent execution of several data localization queries) and (2) reported local processing time since the start. As shown in Algorithm B6, vehicles are for that purpose stored in an updatable priority queue W (initialized in a call to init()) where vehicle v's priority is defined as a tuple of [no. of parallel queries, total local workload]. As shown in line Algorithm 9 of askNewBatch() in Algorithm B6, the vehicle v with the lowest priority is contacted first. After sending a request to v, its priority is updated by increasing the first field of the priority tuple, no. of parallel requests, by one. Likewise at line Algorithm 17 of receiveAndUpdate(), upon

Algorithm B6 BALANCELOAD - asynchronous, static

```
1: procedure INIT()
2:     global W                                          ▷ priority queue
3:     W.insertAll(V, [0, 0])          ▷ initially, all vehicles have same priority

4: procedure ASKNEWBATCH(α,p)
5:     m ← n − |R| − p · (|F| − |A|)
6:     ℓ ← ⌈α · m/p⌉
7:     for 1 ≤ i ≤ ℓ do
8:         if |F| < k then
9:             v → W.getLowestPriority()      ▷ get vehicle with fewest parallel queries, lowest workload
10:            send(q, v)
11:            W.updatePriority(v,[+1, +0])   ▷ increase no. of parallel queries of v

12: procedure RECEIVEANDUPDATE()
13:    r ← receive()
14:    A ← A ∪ {r}
15:    if positive(r) then
16:        R ← R ∪ {r}
17:    W.updatePriority(r.v, [−1, +r.workload])
                          ▷ update priority: decrease no. of parallel queries of sender of r (r.v), increase workload
18: function WAITFORANSWERS(β)                          ▷ same as Algorithm B5
```

reception of an answer r from vehicle v, v's priority is updated by reducing the number of its parallel requests, and increasing the total local workload registered in W for v by the workload transmitted alongside r.

B3.3 Data Localization in the Asynchronous Dynamic Model

The dynamic fleet model introduces vehicles dynamically joining the fleet (which is detected at C) and leaving the fleet (undetected). We describe here adaptations in the presented data localization algorithms to handle both types of events, i.e., vehicles joining and leaving the fleet.

BASEEAGER - asynchronous, dynamic

The algorithm is a straightforward extension of BASEEAGER defined for the dynamic model: all active vehicles get asked upon starting processing a new query at C. Vehicles leaving the fleet will not provide any answer whereas vehicles arriving receive all unresolved queries upon becoming available.

BASELAZY - asynchronous, dynamic

This algorithm could be blocked indefinitely if any of the involved vehicles leave the fleet before the moment when all answers are collected: indeed, the algorithm waits for receiving a negative answer before asking a new vehicle. To deal with

leaving vehicles, we introduce a timer set upon sending a request. A timeout is then considered equivalent to a negative answer and triggers requesting one of the remaining available vehicles; if the timeout vehicle answers later than its corresponding timer, the answer is accepted in case of a yes-answer and ignored in case of a negative one. Contrary to BaseEager, new vehicles may get requested (upon receiving a negative answer or timeout event at C) some time after they become active. However, a new arrival by itself will not trigger directly the transmission of a request, except in the particular case of unresolved queries that have already exhausted the pool of known active vehicles.

BalanceRequests and BalanceLoad - asynchronous, dynamic

Contrary to BaseLazy, the Balance* algorithms as designed for the *static* setting are not at risk of becoming blocked by vehicles exiting the fleet. Indeed, they both already have a mechanism to ask a new batch of vehicles before having answers from all the vehicles that had been requested earlier (through the β parameter, cf. Algorithm B5) and can hence deal with a dynamic fleet where some vehicles leave the fleet. However, over the long run, the estimation of the probability p of answering positively will be less accurate, as vehicles that have left will be excluded from the estimation (only received answers are taken into account); also, if a proportion greater than $1 - \beta$ of the vehicles currently checking requests associated with a particular query leaves the fleet during the algorithm execution, no new batch of vehicles will ever get contacted even though there might be plenty of available vehicles. To circumvent these issues, we also introduce timers in those algorithms: a timeout is equivalent to receiving a negative answer, i.e., a negative answer is added to the set of received answers A, which is used for p's computation and for testing when the β threshold has been crossed. If a vehicle answers positively later than its timer, it is added to the set of known positive answers R; this has no further effect on A, but slightly modifies the calculated value for p as

$$p = \max\left\{\frac{|R|}{|A|}, \frac{1}{|A|+1}\right\}.$$

We note that timers help the estimation p to take into account both the positive answering rate and the fleet churn rate when computing the size of the next vehicle batch to request.

B4 Evaluation

B4.1 Methodology

To investigate the performance of the proposed algorithms, we evaluate them on two large real-world sets of vehicular data. In this section, we first describe in detail the datasets and the induced churn in each of them (see Definition B1) and the experiment setup used for our study. We then present a set of common

queries that will serve to benchmark the different algorithms, including longer-running versions of such queries for our dynamic fleet model. Finally, we show the distribution of data over the fleet and the query answer rates in the studied datasets.

B4.1.1 Datasets

Our evaluation encompasses two datasets (one public, one proprietary) that differ in the number of active vehicles and the rate of churn (see Figure B1), the distribution of data per vehicle (see Figure B3), as well as the types of data included in the dataset.

Geolife Dataset

The first dataset consists of trajectories collected within the scope of the Microsoft Research Asia *Geolife* (version 1.3) project by 182 users over approximately four years [214]. The trajectories were collected from diverse users using different mobile devices and feature predominantly vehicular usage (by car, taxi, or bus). The original dataset consists of 18670 GPS traces containing between 50 and 92,645 records of the form *timestamp (s), latitude (deg), longitude (deg)*. After pre-processing the data, we used 10528 files, each for one day of usage of one user (cf. Figure B1(a) for the number of vehicles over the course of 24h).

Volvo Dataset

The second dataset consists of CAN data and GPS traces from 20 hybrid cars internally collected by *Volvo Car Corporation* [90], [91] in the year 2015. After pre-processing, we generate 3462 trace files, each corresponding to a daily usage of one vehicle (cf. Figure B1(b)). Among the large quantity of CAN data, we have concentrated on two signals, the combustion engine rotation and electric engine rotation. These can be combined, leading to three possible driving modes: ***e**lectric*, ***c**ombustion*, and ***h**ybrid*. Each trace in this dataset hence contains records of the form *timestamp (s), latitude (deg), longitude (deg), driving mode (e/c/h)* (cf. Figure B1(b) for the vehicle number over the course of 24h).

Vehicles leaving and joining the fleet in the datasets

$Churn_{\Delta}(t)$, measuring the fraction of vehicles leaving the fleet within a predefined time interval Δ, influences how fast queries get resolved (see Definition B1). In the studied datasets, the churn is evaluated to be between 2% (for $\Delta = 30$ seconds) and 38% ($\Delta = 15$ minutes), see Figure B1(a) and Figure B1(b). In a general sense, churn not only describes vehicles leaving the fleet while the latter is processing requests but also associates with communication issues due to the high node mobility, with many vehicles featuring intermittent short activeness periods, typical of dense urban driving. A non-negligible churn causes problems to data localization algorithms as explained in Section B3.3. In the majority of our experiments (Section B4.1.4 to Section B4.2.4), there is negligible churn

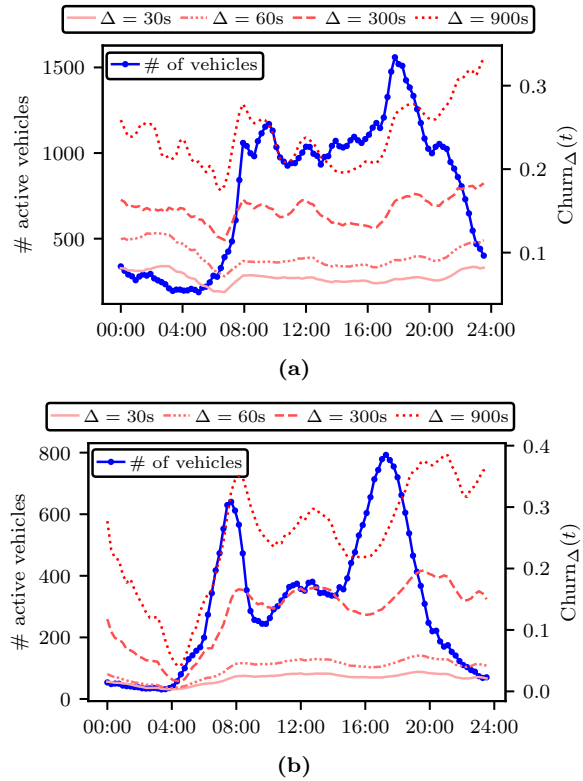


Figure B1: Number of active vehicles and churn during one day in the (a) *Geolife* and (b) *Volvo* dataset (see Definition B1 for a formal definition of *churn*).

in the fleet during query execution when regarding the timescale for query resolution (with queries lasting only up to 30s, and $0.02 \leq \text{Churn}_{30s} \leq 0.08$ as shown in Figure B1(a), Figure B1(b)). Longer queries, subject to longer churn intervals, are studied in Section B4.2.5.

Dynamic changes to the active fleet pool are also based on arriving vehicles. While vehicles joining the pool do not alter the execution of the requests being currently processed by the fleet, new vehicles support the execution of the running queries when the number of vehicles with positive answers to a query is scarce or declining due to non-negligible churn (exacerbated for the *Volvo* dataset with its lower active vehicles count).

B4.1.2 Experiment Setup

We will present here the components and key settings of the evaluation of our proposed algorithms, involving the adaptation of real-world datasets and parameters.

Query response time calculation

To evaluate our algorithms, we define 15 queries to be run locally on the vehicles (presented in Section B4.1.3). The requests are programs written in Python that are transferred to the vehicle via mobile broadband communication, then executed *on-board* the vehicle over their already stored data (1 day each); size(q) denotes the amount of code and extra data[(B1)] that needs to be transferred from C to each vehicle in order for the latter to be able to process $r(q)$ on-board. The elapsed time $R(v, q)$ (in milliseconds) needed between the coordinator sending a request message $r(q)$ for query q to a vehicle v and the reception of the corresponding answer is approximated as

$$R(v, q) = T_l + \frac{\text{size}(q)}{T_d} + T_p(v, q) \tag{2.4}$$

where T_l is a round-trip latency for wireless communication, T_d is the wireless link data rate, and finally $T_p(v, q)$ is the time needed by the vehicle to decide if it can answer positively to $r(q)$ or not. The transmission time of the answer, considering that the answer is of constant and small size (for a yes/no reply and a constant amount of additional information such as the vehicle id, the time it took for the processing, etc.), is neglected here (it can be accounted as part of T_l).

Resolution time of concurrent queries

The experiment process is done as follows. The coordinator node C receives a certain number of queries in a random uniform order and starts the batch of sending requests to vehicles in the same order as the queries' arrival times. The queries are then resolved in parallel by the vehicles and C reacts to each message reception by either just updating its internal statistics for the corresponding query q or by spreading the request $r(q)$ over the fleet to a new set of vehicles. As introduced in Section B2.1, vehicles possess a *task queue* processed in FIFO order. This approach simplifies the vehicles' internal computing architecture and is well suited in situations for which the remaining computing resources on-board the vehicles (if any) can be used to process security-sensitive applications. A vehicle v hence starts processing a request as soon as v is done with the processing of its already queued tasks. When considering multiple queries concurrently processed at C, the reception time $R'(v, q_k)$ of v's answer to the request $r(q_k)$ corresponding to the k-th received query q_k at v is obtained as

$$R'(v, q_k) = \max\{R'(v, q_{k-1}) + T_p(v, q),\, t_k + R(v, q)\} \tag{2.5}$$

where $t_1 < t_2 < \cdots < t_k$ indicate the sending times of requests $r(q_1), \ldots, r(q_k)$ to vehicle v, and with $R'(v, q_1) = R(v, q_1)$ where $R(v, q)$ is calculated using Equation 2.4.

[(B1)] For example, GPS positions of Points of Interests (POIs) such as parking lots or fuel stations.

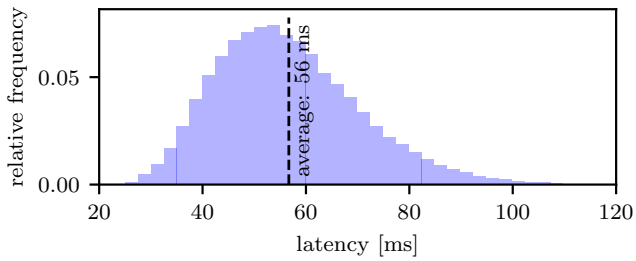


Figure B2: Distribution of wireless round-trip latencies T_l (modeled after [146]).

Real-world values used for the parameters

In our set of experiments, we have set $T_d = 10$Mb/s, which is within current 4G/LTE download rates[(B2)] (similar results are obtained using 5G parameters). To model a non-deterministic but realistic 4G round-trip latency T_l, we sample T_l randomly from the Gamma distribution shown in Figure B2, as modeled after the results from a study of 4G latencies across several mobile carriers in the UK [146]. To have a fair estimation of $T_p(v, q)$, we have computed all queries on a vehicular processing unit representative [90], [91]: an ODROID-XU4 single-board computer to approximate the limited processing headroom of a vehicle, equipped with a Samsung Exynos 5422 (Cortex-A15 2.1GHz Quad-Core and 1.4GHz Quad-Core CPUs) and 2 GB of LPDDR3 RAM at 933 MHz. We then use the computed time measured on the vehicular stand-in as $T_p(v, q)$ for every possible vehicle v and query q. Based on the measured transfer time (through an Ethernet link with software-capped bandwidth to $T_d = 10$Mb/s), size$(q)/T_d$ expressed in ms is very well approximated by the size of data to transfer expressed in Kb.

B4.1.3 Selected Data Localization Queries

In this subsection, we present our selection of data localization queries used for the static and dynamic fleet scenarios. Please note that these queries are tailored to the two datasets/fleets employed, each of which has a known (geographic) focus. In the general case, basic a-priori knowledge, e.g., vehicle type or region (which can be assumed to be known to the vehicle manufacturer), can be used to select a subset of a fleet before deploying the actual query over the now filtered fleet.

Queries for the static fleet

We introduce here a set of 15 queries, representative of possible vehicular analysis tasks. The queries match typical interesting events occurring in Vehicular Networks [101] (driving close to POIs such as parking spaces, detecting traffic jams, etc.), thus giving meaningful insights into the fleet's behavior. They were chosen to represent different requirements (on time interval, queried

[(B2)] To take into account packet losses, T_d is chosen inferior to typical broadband bandwidth.

Table B1: Selected query conditions and their parameters (QR = share of positive answers over the dataset, G = *Geolife* and V = *Volvo*).

Key	Time span	Size (Kb) in G/V	Condition to fulfill "$q.P$" for G	for V	QR in G	QR in V
Q_1	8-12	0.3	At least 1 record during the time span		56	60
Q_2	0-24	7.5/19.9	Driven within *50m* of any parking lot	*25m*	42	43
Q_3	17-18	1.3	Continuous records *with* ζ_G[a]	*with* ζ_V[a]	29	28
Q_4	0-24	0.5	Driven through City area	*with* ζ_V		18
Q_5	17-18	1.1	Maximum speed reached *over 89km/h*	*over 99km/h*		12
Q_6	0-24	1.6	Instant speed over 42km/h *for 10min*	*for 18min*		8
Q_7	0-24	1.4	Driven in Downtown *with* ζ_G consecutive records	*with* $\zeta_{V'}$[a]		5
Q_8	17-18	0.8	Passed by *City area*	*Downtown area*		4
Q_9	12-13	0.7	Stayed in City area for all time span			1
Q_{10}	0-24	3.8/7.7	*Stopped* at any gas station for a short duration[b]			1
The queries below are only defined for the *Volvo* dataset.						
Q_{11}	0-24	4.8	*Combustion* engine used less than 10% of the time			34
Q_{12}	0-24	5	Driven on *electric* mode only outside City area			25
Q_{13}	0-24	4.4	Driven using *hybrid* mode for 10min			10
Q_{14}	0-24	6.6	Instant speed on *electric* mode reached over 100km/h			7
Q_{15}	0-24	4	Passed by 3 distinct electric vehicle charging stations on *electric*			4

[a] $\zeta_G = (80, 5)$, $\zeta_V = (85, 10)$, $\zeta_{V'} = (300, 10)$, where $\zeta = (\tau, \delta)$ requires at least τ consecutive measurements spaced at least δ seconds apart.
[b] For *Geolife*, we require ζ_G records within 50m of a gas station; for *Volvo* the vehicle must stop (speed = 0km/h) for 10min within 20m of it.

sensors, geographic constraints, sampling constraints, etc.). They furthermore have distinct positive answer rates ranging from about 60% to about 1%. Table B1 presents (cf. Section B2 for notations) the query q's key (Q_1 to Q_{15}), the time interval $t_{start} - t_{end}$ given in hours, size(q) given in Kb, the description of the condition $q.P$, and the average answer rate QR (rounded to closest percentage) for *Geolife* and *Volvo* datasets. The parameters of the first 10 queries have been slightly tuned between the two datasets (in Table B1 the additional column for $q.P$'s description indicate differing parameters in the query's condition in *Volvo*) so that each query in both datasets has a similar fraction of positive answers. Recall that size(q) corresponds to the size of the program plus the extra data required to check $q.P$, cf. Section B4.1.2. Of the queries, 10 are run over both datasets whereas 5 additional queries focus on signals only contained within the *Volvo* dataset. Two geographical zones are defined for both datasets: City is the area of a large city chosen within the dataset and Downtown is a sub-area within City thought of as its heart. In our experiments, if not stated otherwise, all queries will require $n = 50$ answers to get resolved. Setting an adequate value for the parameter n is a non-trivial task that is both query- and data-dependent and is linked to the post-treatment of the vehicle selection process and the end-application. For the case of statistical estimation of the true answer rate QR, The impact of the choice of n with the presented algorithms is explored thoroughly in Section B4.2.4, where the value $n = 50$ is shown to provide a good trade-off between estimation accuracy and excessive vehicle involvement over the queries analysed in this work.

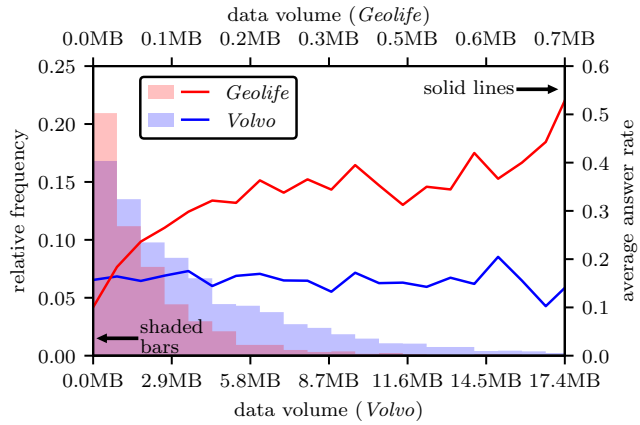


Figure B3: Left axis: Histogram of data volumes per car over each respective dataset. Right: Average answer rate for a vehicle with a certain amount of data. Results shown for the *Geolife* (red) and *Volvo* (blue) datasets.

Queries for the dynamic fleet

As mentioned in Figure B4.1.1, short queries (in terms of resolution time) entail a similar behavior in a dynamic fleet as the fleet remains stable during the time used to resolve the query. All queries defined so far fall in this category, as most of the time, they get resolved in less than one second – whereas the churn for 30s is below 5% of vehicles (cf. Figure B4.1.1). To be able to observe differences in the algorithms' behaviors, we introduce "long" versions of the ad-hoc queries previously introduced and described in detail in Table B1. The long versions are obtained by multiplying both the transfer time of the requests and the time needed to process them by a constant of 1000, for a realistic distribution of answering times representative of a fleet with a higher amount of local data or heavier computational tasks used for queries' conditions.

B4.1.4 Distribution of Data and Query Answers Rates

The average answer rates over all queries as well as the distribution of the data volumes are presented in Figure B3 for the *Volvo* and *Geolife* dataset; the x-axes range over data volumes in MB within *Volvo* (lower axis) and *Geolife* (upper axis) datasets. For *Geolife*, the average query answer rate (red line) appears to be positively linked to the data volume (shaded red bars); thus, vehicles with larger amounts of data will have a higher chance to answer requests. For *Volvo* (blue line), the average query answer rate is almost flat, which indicates that vehicles with a large amount of data (shaded blue bars) are roughly as likely to answer "yes" to a request as vehicles with only little data. Concerning the distribution of data volumes among the fleet (shaded bars), the *Volvo* dataset presents a significantly longer tail, indicating that inter-vehicular differences in data volume are greater.

B4.2 Evaluation Results

We show in this section the experiments' results. To compare the performance of the different algorithms, we will use the evaluation metrics defined in Section B2.4, namely the query resolution time and the fleet workload. To show the results, we will frequently use violin plots, which indicate the median of a distribution with a horizontal bar, while the distribution itself is shown vertically in shaded color.

B4.2.1 Parametrization of the Algorithms

To choose well-fitting parameters for our evaluation, we explore the parameter space for BalanceRequests in the *Geolife* and *Volvo* dataset. We run 10000

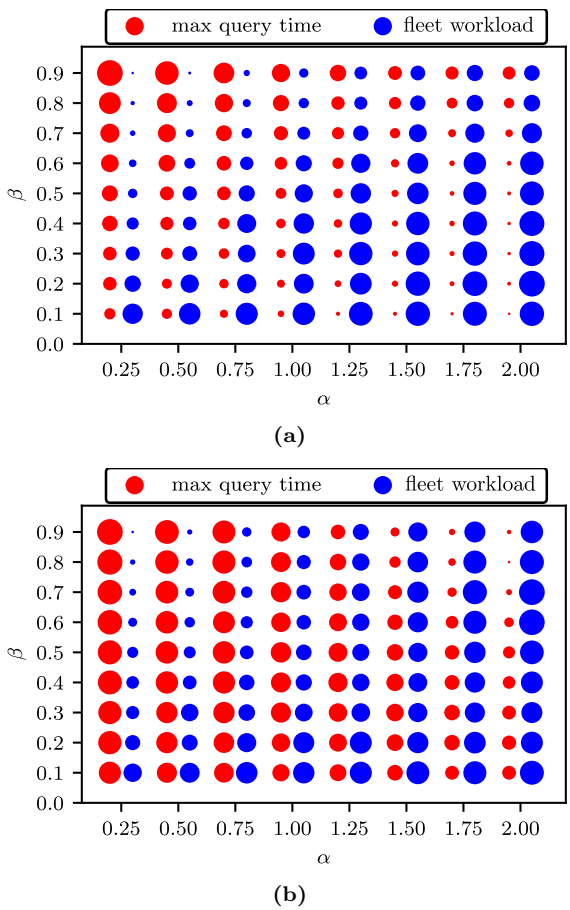


Figure B4: Maximum query resolution time and fleet workload *(static model)* needed to resolve all queries over the *Geolife* and *Volvo* datasets for BalanceRequests for different α, β. Circle size scales with maximum query resolution time (red) and fleet workload (blue), respectively.

times the query sets with different values for the parameter α (proportion of vehicles to ask; higher value translates to asking more vehicles) and β (fraction of vehicles to wait before asking next batch; higher fraction translates to longer waiting time between two request batches). For each run, we measure the time needed to resolve all queries (i.e., the maximum query resolution time among the query set) and the fleet workload and present them on a 2D plot in Figure B4 (note that absolute values are given in Figure B5).

Based on the fleet workload (blue) displayed in Figure B4, in both datasets, for lower values for β and higher values for α, more vehicles than necessary tend to be requested while not waiting for everyone's answer before requesting a new batch of vehicles. The consequence in this setting is on the one hand a high analysis cost, as more vehicles participate in the queries resolving task, but on the other hand, the resolution time is relatively lower than other configurations of (α, β). Focusing on the maximum query resolution time (red), the situation is different between the *Geolife* or *Volvo* dataset: When vehicles tend to answer "no" because of lack of data (as for the *Geolife* dataset, cf. Section B4.1.4), hence responding much quicker than positive vehicles, and β is rather small, the estimation p (cf. Algorithm B3) of positive answers will be too low; then (as β is small) many vehicles are requested rapidly in the first few rounds. The consequence is a shorter resolution time but higher fleet workload, as seen in the *Geolife* experiments. On the contrary, if the data is distributed over the fleet more fairly (as for the *Volvo* dataset, cf. Section B4.1.4), and when β is low, there will be a bias towards positive answers with queries that require a full data scan before they can be answered negatively, whereas vehicles answering positively need only find the first matching record(s). The consequence is that p becomes an overestimation of the real fraction of yes-answers and one observes a succession of small batches of vehicles being requested, as observed in the *Volvo* experiments. When β approaches 1, the estimation p becomes unbiased and better trade-offs are obtained; however, note that a high β is impractical for longer queries, as is shown in Section B4.2.5. For the remainder of this section, we set $\alpha = 1.25$ and $\beta = 0.7$ as these values present a suitably balanced trade-off between the two measured performance metrics over both datasets; other nearby values for (α, β) produce similar results that only slightly advantage one metric over the other, as explained above.

B4.2.2 Comparison of the Algorithms in the Static Model

To give a general idea of the resolution time for the different queries, we present quantitative results in Figure B5 (an intra-algorithms comparison is done in Figure B6): the query resolution time is measured over 10000 experiment repetitions consisting in resolving all 10 (*Geolife*, (a)) / 15 (*Volvo*, (b)) queries arriving in random order; for the *Volvo* dataset, Q_9 and Q_{10} have been removed here and for all following experiments as all vehicles end up being contacted (there are fewer than 50 positive answers in this case, violating the assumption $k_q > n$ from Section B2.1). The main findings to note are: BaseEager's and BaseLazy's resolution time varies clearly depending on the queries' answer rate (lower answer rate QR is associated with larger resolution times, see Table B1

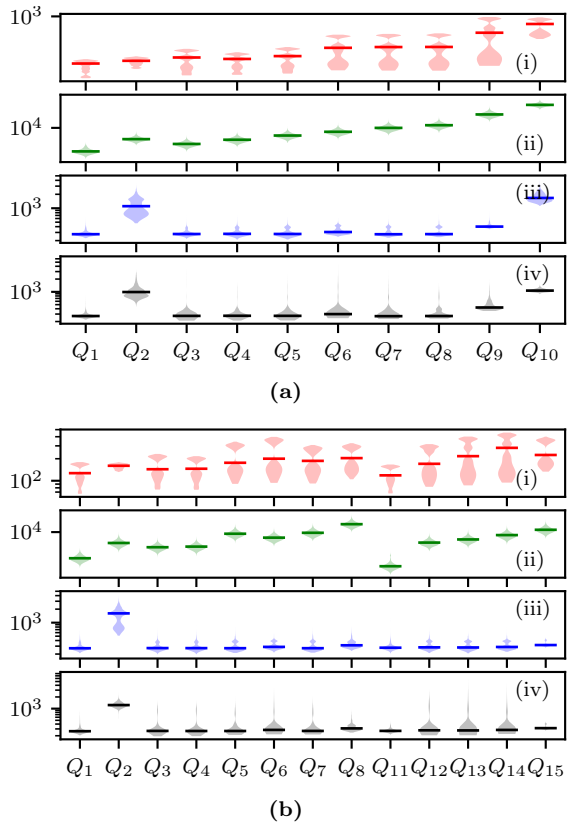


Figure B5: Query resolution time (in ms) *(static model)* for all valid queries executed over the (a) *Geolife* and (b) *Volvo* dataset for (i) BaseEager, (ii) BaseLazy, (iii) BalanceRequests, (iv) BalanceLoad.

for QR per query), while BaseLazy is one to two orders of magnitude slower; and BalanceRequests and BalanceLoad present similar query resolution times that do not vary significantly with the queries' answer rate (except for Q_2 and Q_{10} [*Geolife*]). Also, note these computationally heavier queries Q_2, Q_{10} (requiring to check spatial proximity to multiple points of interest) get resolved significantly slower than lightweight queries. BaseEager shows large variations in resolution time for the same algorithm and query because, contrary to all other algorithms, the algorithm itself is purely deterministic and highly dependent on the order in which queries arrive: indeed, if a "heavy" query is sent first to every vehicle, all the nodes will need to process it before moving on to the next query (cf. the FIFO task queue as described in Section B2.1), potentially slowing down subsequent lighter queries (this also explains why the balanced-algorithms may outperform BaseEager by contacting smaller subsets of vehicles, see Figure B10).

As a summary, Figure B6 presents the query resolution time (left side) and fleet workload (right side) over all queries for the four algorithms relative

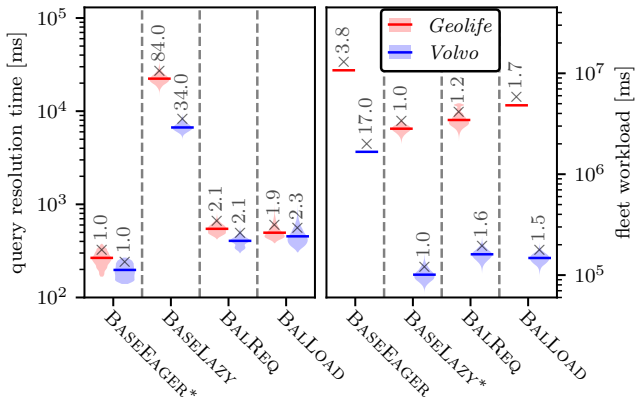


Figure B6: Query resolution time (left) and fleet workload (right) *(static model)* of the four algorithms for the *Geolife* (red) and *Volvo* dataset (blue). The starred algorithm is the respective baseline, the y-axis is logarithmic to suit the different scales.

to the average resolution time of BaseEager and the average fleet workload of BaseLazy, respectively (marked by stars). The query resolution time is almost two orders of magnitude higher for BaseLazy (in the *Geolife* dataset) than for BaseEager, whereas the Balance- algorithms are almost as fast as BaseEager, which shows the best resolution times in both datasets. For the fleet workload, BaseLazy outperforms BaseEager by a factor of up to 17. The Balance* algorithms perform again well in this metric on both datasets, having an average cost close to the baseline.

Finally, BalanceRequests's fleet workload shows the dependence on the distribution of data in the fleet: with skewed data (as in the *Geolife* dataset), it outperforms BalanceLoad by a margin of 40%, whereas in a uniformly spread dataset (e.g., *Volvo*) it performs marginally worse.

B4.2.3 Fairness of the Algorithms

The presented algorithms distribute clearly differently the workload over the vehicles. We measured the standard deviation σ_{load} of the local workloads *between* the vehicles with non-zero workloads, for executing all queries (cf. Section B4.1.3) over 10000 experiments and for both datasets, presented by Figure B7. Low values of σ_{load} indicate that all vehicles have a similar workload, and vice versa for high σ_{load}. In BaseEager, the workload is distributed deterministically as every vehicle checks every query (even though the execution order may vary in different runs), and thus the value for BaseEager results only from vehicles needing different times to execute all queries. The inter-dataset differences may be explained by the fact that *Geolife* exhibits larger variance in the average answer rate between vehicles, cf. Figure B3. All other data localization algorithms show a smaller spread of the workload, hence a fairer distribution, as more vehicles have similar workloads. BaseLazy provides

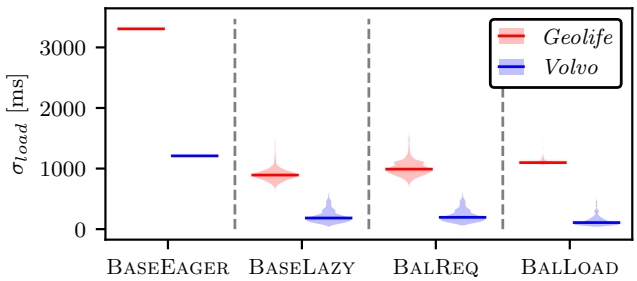


Figure B7: Standard deviation σ_{load} of local workloads *(static model)* between vehicles over both datasets and 10000 experiments.

the fairest outcome in this sense in the *Geolife* dataset, closely matched by BalanceRequests and BalanceLoad, while the latter provides small improvements over the three in the *Volvo* dataset.

B4.2.4 Estimation of the Fraction of Yes-Answers

In all previous experiments, the number n of required answers was set to 50. This section now investigates how this parameter influences the outcome of the different presented data localization algorithms. Recall that the number of required answers allows one to select a fixed number of vehicles from the fleet satisfying the query's condition for further analysis. One may estimate in this fashion the true fraction QR of vehicles satisfying the query in the full fleet, with a higher number of required answers providing intuitively a better estimation of that fraction. The estimation is given by n/m, where n yes-answers have been collected over m total received answers. Here, the validity of the aforementioned intuition will be investigated. Figure B8(a) presents the average absolute error $\Delta QR = |QR - n/m|$ on the estimation of yes-answers among the fleet for the first five defined queries with $QR = 56/60\%$, $42/43\%$, $29/28\%$, 18% and 12%, respectively (*Geolife/Volvo*, cf. Table B1 for details about the queries), and n ranging from 10 to 175 required answers. For each n and each algorithm, 10000 experiments were conducted where all 5 queries are being resolved in parallel. Then, for each experiment and each query, the share n/m of yes-answers provided by the algorithm at the moment that the particular query is resolved is recorded. Since BaseEager asks every vehicle in the fleet, the estimation is provided based only upon the fastest vehicles to answer and ends up providing the least precise estimation of all tested algorithms. On the contrary, BaseLazy, by asking one vehicle at a time chosen randomly upon receiving negative answers, bases its estimation on a purely random pool of vehicles, hence providing the best estimation unbiased by the time vehicles require to answer the query. In between, BalanceRequests and BalanceLoad provide reasonable trade-offs; as both base their estimation on the first 70% of vehicles to answer a particular query (as $\beta = 0.7$), they both feature bias as BaseEager, but to a lesser degree. The main difference between the two algorithms is that BalanceLoad introduces another bias

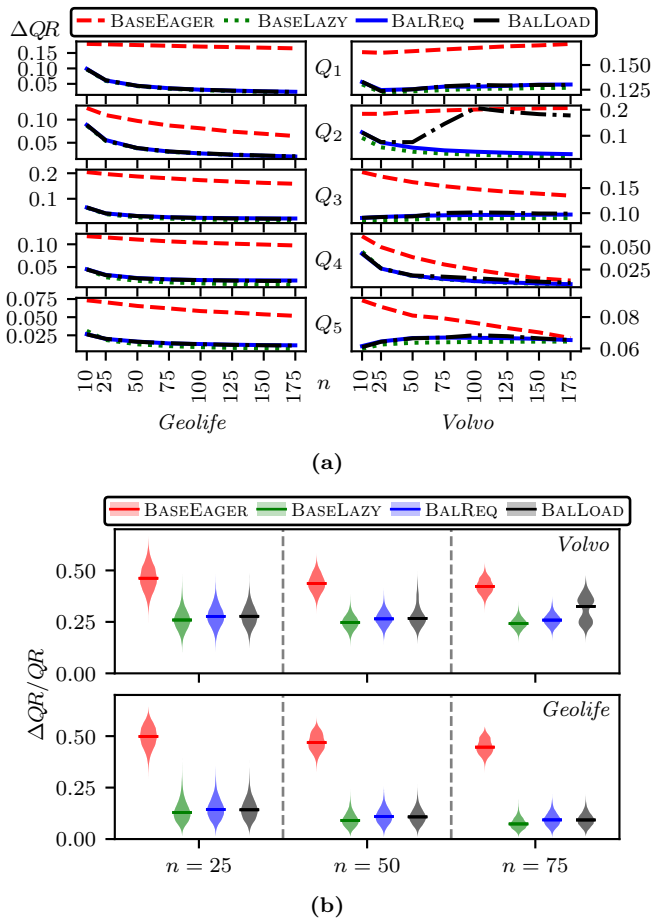


Figure B8: (a) Average absolute error ΔQR and (b) relative error $\Delta QR/QR$ of the estimation of yes-answers *(static model)* for queries $Q_1 - Q_5$ with different data localziation algorithms over both datasets.

on top of using the 70% fastest vehicles, which is selecting vehicles with a current lower load rather than random ones as in BalanceRequests; this additional bias seems strongest in Q_2 in the *Volvo* dataset. On most queries, the balanced algorithms perform nearly as well as BaseLazy. We note that they present almost identical estimations except for query Q_2, where BalanceLoad, prioritizing spreading the queries fairly among the fleet, under-performs in the *Volvo* dataset. Figure B8(b) summarizes the estimation performance of all four algorithms on both datasets by presenting the average relative error $\Delta QR/QR$ of the estimation of yes-answers for the 10000 experiments. The advantage of the introduced algorithms is clear: they provide mostly good estimates independently of the required number of answers, especially considering the low values of n compared to the size of the fleets (*Geolife*: 10528; *Volvo*: 3462), while the disadvantage of the secondary bias of BalanceLoad becomes

apparent again in the *Volvo* dataset.

B4.2.5 Comparison of the Algorithms in the Dynamic Model

Recall that in the dynamic model, vehicles may join and leave the fleet at any time during a query's resolution. For the churn values to be in line with those observed in real setups (cf. Section B4.1.3), we have modeled the arrival and departure of vehicles during a time interval by using real-world traces: in the following experiments, the set V_t representing the fleet at time t consists of all vehicles that have records within 7.5s of t. Furthermore, here long queries defined in Section B4.1.3 have been used where the multiplicative factor has been set to 1000. This shifts the fastest vehicles from answering within milliseconds to seconds and the slowest from a few hundreds of milliseconds to minutes (on selected queries); similarly, the transfer time of 1-20ms becomes 1-20s (the equivalent of 1-20 MB of data having to be transferred per query). The timeout (introduced in Section B3.3) is set to 100s and the number of answers required per query is set to $n = 50$ as in previous experiments.

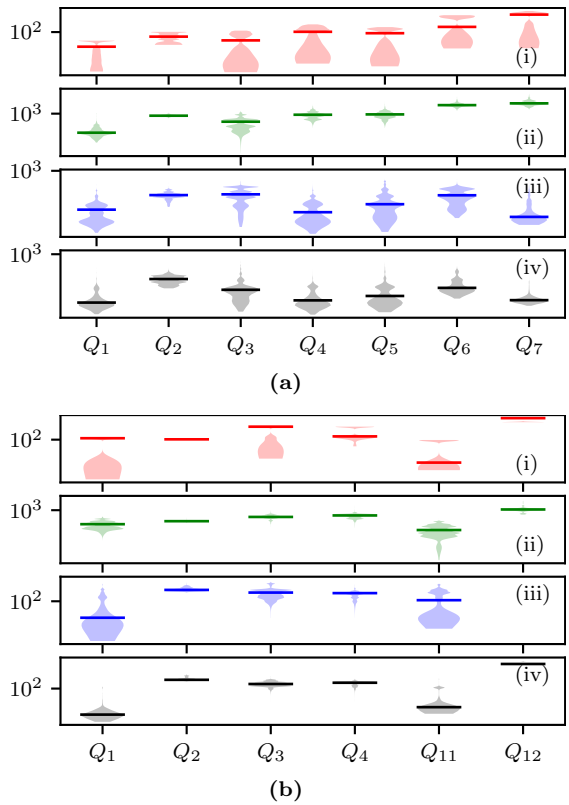


Figure B9: Query resolution time (in s) *(dynamic model)* and all valid queries executed over the (a) *Geolife* and (b) *Volvo* dataset for (i) BaseEager, (ii) BaseLazy, (iii) BalanceRequests, (iv) BalanceLoad.

Figure B9 presents the query resolution time in the dynamic model following the same conventions as Figure B5. We use in the experiments a query batch of 7 queries (*Geolife* dataset) and 6 queries (*Volvo*) with a starting time of 18:00; the remaining other queries were discarded as not solvable considering only vehicles active past that point in time. The main outcomes in regards to the adaptation of the algorithms to dynamicity are as follows: (i) BaseEager's and BaseLazy's resolution time is less dependent on the queries' positive answer rate; (ii) BalanceLoad performs similarly to BalanceRequests with a slight improvement thanks to spreading the requests over more vehicles, which then decreases the chance that a vehicle leaves the network before having emptied its local request queue; and (iii) queries that require new arrivals to get resolved, such as Q_{12}, display high resolution times regardless of the spreading algorithm used (however, we note that in these situations, BalanceRequests is more likely to fail to collect enough answers, as it does for Q_{12} in Figure B9 b).

Figure B10 shows, as a summary, the average query resolution time and fleet workload over all selected queries[(B3)] of both datasets for each algorithm relative to BaseEager and BaseLazy, respectively (in a similar fashion as Figure B6). Contrary to the static setting, where the best-performing of the Balance- algorithms varies depending on the distribution of data over the fleet (cf. Section B4.2.2), in a dynamic environment, BalanceLoad clearly performs best. BalanceRequests displays overall favorable trade-offs, performing close to each baseline, but slightly slower and with a higher cost than BalanceLoad. The latter performs close to or better than BaseEager in terms of time, and better than BaseLazy in terms of load. This is the consequence of the way the vehicles are selected in the algorithm; those with the lowest current local load are requested first (hence favoring freshly arrived vehicles), which in our datasets

[(B3)] All queries described in Figure B9, except for *Volvo* where Q_{12} has been excluded from the resolution time plot, as it did not always terminate.

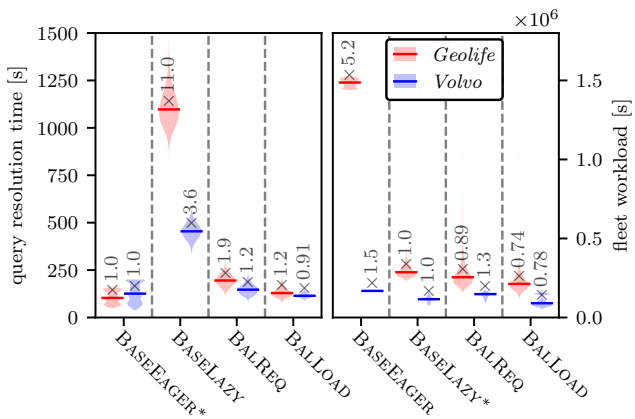


Figure B10: Query resolution time (left) and fleet workload (right) *(dynamic model)* of the four algorithms for the *Geolife* (red) and *Volvo* dataset (blue). The starred algorithm is the respective baseline.

biases the selection process towards vehicles with higher chances of answering positively or vehicles answering faster (see discussion in Section B4.2.1). A clustering of relevant data in those vehicles may further exhibit the causes of BalanceLoad's higher performance.

B4.2.6 Summary of the Results

Table B2 summarizes the average resolution time and relative fleet workload (compared to BaseEager) over all queries for all algorithms and datasets in the static and dynamic fleet model. On average, the proposed algorithms resolve queries up to 40 times faster than BaseLazy while consuming only 1/3rd of the resources of BaseEager (BalanceRequests, static model, *Geolife*).

The presented solutions (a well-tuned BalanceRequests and BalanceLoad) provide substantially improved trade-offs of query resolution time versus on-board workload compared to baseline solutions, and allow tuning between the tradeoffs by varying the estimation of required vehicles to ask in the next iteration (via the parameter α) and the waiting times for slow-processing vehicles (via β). Furthermore, a query's resolution time in the proposed algorithms is shown to not be negatively impacted by a low positive answer rate among the fleet.

BalanceLoad, presenting shorter resolution time and slightly larger fleet workload, produces a workload more fairly spread over the vehicles; however, it may provide a less accurate estimation of the fraction of positively answering vehicles once queries are resolved. BalanceRequests provides the most balanced trade-offs overall, performing almost as good as each baseline solution both when considering a uniform distribution of positive answers (*Volvo* dataset) or a skewed distribution (*Geolife* dataset). Finally, BalanceLoad is overall more suited to "dynamic" scenarios, i.e.,when the queries require long enough processing times for the fleet churn to become noticeable.

Table B2: Summary of the results (average over all queries) for both datasets, all algorithms, and static & dynamic models.

	Algorithm	**Resolution time (s)**		**Fleet workload (s)**	
		Geolife	*Volvo*	*Geolife*	*Volvo*
static	BaseEager	0.27	0.20	10715	1667
	BaseLazy	22.65	6.75	2823	103
	BalanceRequests	0.55	0.40	3616	162
	BalanceLoad	0.52	0.48	4816	148
dynamic	BaseEager	105	204	1484451	168280
	BaseLazy	1105	556	293245	115858
	BalanceRequests	197	148	281650	141771
	BalanceLoad	132	220	227068	101282

B5 Related Work

Having studied in this work the problem of how to localize, efficiently and in a distributed manner, relevant data in a vehicular fleet for analysis applications, in the following paragraphs we discuss work about topics that associate with or have similarities to the problem.

The traditional approach to query a set of vehicles has been through SQL-inspired languages [97], [204], [210] to process continuous queries on live vehicular sensors' data. Two main differences with the current work are that in previous works (i) "queries" were usually initiated by vehicles themselves (e.g., [5], [51], [120], [123], [124], [211]) and (ii) the full fleet was queried upon receiving new queries (as in [121]), contrary to our work in which a known and fixed set of *general* queries is deployed from a centralized point to the fleet and only some vehicles in the current fleet may have relevant data to answer the queries. Also, many works in the field are based on an advantageous usage of geographical properties of the distribution of Road Side-Units (among others [5], [51], [118], [120]), whereas our work is only based on the already widespread mobile broadband infrastructure as well as data analysis capabilities already in place at car manufacturers' data centers. Query-answering mechanisms for Vehicular Networks in the literature also predominantly concentrate on using the architecture of the network (for instance using pre-existing P2P approaches, as in [116], [129], [197] or 2-tier architectures [39], [186]) to resolve the query. In this work, we do not presume any connections between vehicles; this positions our work in readily deployable technologies on modern vehicles. A querying approach for vehicle selection was recently studied in [53] in which a request is sent to all available vehicles to detect candidates to participate in Federated Learning. The vehicles send updated responses to the query over time as they are collecting new data, and eventually a subset of the vehicles that answered positively is chosen. In contrast, our algorithms attempt to limit the number of vehicles that are queried for data, thus allowing for more concurrent queries, while novel data discovery is only supported via new queries.

The problem of localizing the relevant data or "data localization" features many similarities with the concept of data aggregation in wireless sensor networks [92], [113], [117], [148], [164]. Usual aspects of data aggregation that differ from data localization include a continuous aspect (rather than an on-the-fly query approach for data localization) and dissemination of information to nearby nodes (rather than to a single sink node for data localization). Since the focus taken here is on the localization of data, approaches for efficient data gathering [58], [90], [91] and aggregation do well complement the initial localization phase. The way our algorithms have been designed also relates to the large field of information gathering (see [36] and references therein). In both concepts, online decisions are iteratively taken to gain knowledge of a hidden state. For instance, in [37], the authors design near-optimal algorithms to pick the right set of tests in order to maximize the *value of information*, with applications to medical diagnosis and troubleshooting. Note such approaches can be used to design the right set of queries (similar to the aforementioned tests) to resolve a particular task whereas our work concentrates on how to

efficiently and distributively resolved those queries.

The fundamental opposing metrics studied in our paper that are the time to resolve the queries and the computational overhead induced on the fleet are similarly observed in the field of job scheduling in distributed computing. Parallel and redundant job execution can decrease job execution times in heterogeneous environments (e.g., through speculative scheduling in *MapReduce* [208]), at the cost of increasing contention and overall workload [6]. In contrast, in our work parallel execution is always required, and no node is a priori known to be fundamentally able to fulfill a given task.

In vehicle data analysis, privacy aspects are important when dealing with for example location-based services [42], [95], [202] and privacy-preserving cloud-based query processing [118]. We suggest that our work, by allowing to check whether a certain number (chosen by the analyst) of vehicles meets a given condition, can complement applications where privacy is supported by aggregating data from many sources.

B6 Conclusions

This work proposes two distributed algorithms for data localization in Vehicular Networks. To the best of our knowledge, this paper is the first to propose a data localization mechanism over a Vehicular Network through request spreading, focusing on acquiring only a limited number of answers from the fleet and considering as a performance metric the computing workload of the vehicles. The focus lies on the vehicle selection phase necessarily performed prior to data gathering over large vehicular fleets, typically for selecting vehicles that triggered a particular condition or event [90], [91], [119]. As this work also shows, this vehicle selection mechanism can be used as-is for estimating the occurrence of particular events in the vehicles' recent data while incurring low overhead on the Vehicular Network as a whole. The proposed algorithms balance (i) the overall time needed to identify a subset of vehicles holding relevant data and (ii) the local computational overhead each vehicle pays to check whether a set of properties hold for its data. As shown with analytical argumentation and experimental evaluation, conducted with real-world data traces, the algorithms provide means to tune the trade-off between (i) and (ii) in interesting ways. In particular, it appears that it is possible to significantly reduce the query resolution time, with only a small extra load imposed on the vehicles, compared to the baselines that can optimize only one of these metrics (achieving for example up to 40 times faster resolution while saving more than 65% of the computing resources). These results indicate that the adoption of a data localization phase prior to the execution of additional analysis steps for example in a Federated Learning scenario can occur with little overhead with respect to time and computing resources, thus enabling better learning outcomes for a comparably small price. Furthermore, our results show that the distribution of the work to the vehicles can happen fairly, even for skewed data distributions, to alleviate the risk of overloading individual vehicles. This work sets the basis for several paths to investigate in the future. One avenue is the

porting of the proposed algorithms to V2V [74] rather than centralized V2I setups. A second direction is to explore how our algorithms can be integrated within existing simulators (e.g., with a traffic and/or network simulator) to produce richer simulation environments for benchmarking smart analysis in Vehicular Networks. Lastly, investigating the use of correlations between queries could be a promising way of efficiently selecting those vehicles for a query that have answered positively to a similar query in an earlier execution by adaptively changing the parameters of our algorithms during a data localization query's resolution.

Appendix

A preliminary formulation of the problem was presented in [59]. The present article builds on those test results and presents an extensive study that includes a detailed problem formulation, as well as varying system models and parameters along with algorithmic designs for them. For comparison, we list here the main novel contributions of the present work:

1. the system model is made more realistic by introducing a dynamic fleet model where vehicles can leave and join at any time (Section B2.2);
2. the data localization algorithms presented here are enhanced to adapt to changes in the set of available vehicles (Section B3.3), in accordance with the new system model;
3. the evaluation has been updated with use-cases that account for the new system model (Section B4.2.5);
4. the evaluation is extended with a thorough comparison of the algorithms' behaviour, using a larger set of performance metrics, as well as enhanced experiment repetitions for higher statistical certainty, and a more extensive analysis of the parametrization of the proposed algorithms (Section B4.2.3, Section B4.2.4);
5. the study includes more realistic modeling of the communication delays (Section B4.1.2); and lastly
6. the evaluation framework and algorithms are openly published[(B4)] to enable replicability of our experiments as well as to spark further research.

[(B4)] All code and detailed useage instructions are available at `https://github.com/dcs-chalmers/dataloc_vn`.

Chapter C

Forward Provenance for Data Selection in Stream Processing

Dimitris Palyvos-Giannas, **Bastian Havers***, Marina Papatriantafilou, Vincenzo Gulisano

*The first two authors contributed equally to the publication this chapter is based on.

The following is an adapted version of the work published in *Proceedings of the VLDB Endowment, Vol. 14 (3), p. 391-403*, as *"Ananke: A Streaming Framework for Live Forward Provenance"*. Any changes serve only to retain the consistency of this thesis.

Abstract

Data streaming enables online monitoring of large and continuous event streams in Cyber-Physical Systems (CPSs). In such scenarios, fine-grained backward provenance tools can connect streaming query results to the source data producing them, allowing analysts to study the dependency/causality of CPS events. While CPS monitoring commonly produces many events, backward provenance does not help prioritize event inspection since it does not specify if an event's provenance could still contribute to future results.

To cover this gap, we introduce *Ananke*, a framework to extend any fine-grained backward provenance tool and deliver a live bipartite graph of fine-grained forward provenance. With *Ananke*, analysts can prioritize the analysis of provenance data based on whether such data is still potentially being processed by the monitoring queries. We prove our solution is correct, discuss multiple implementations, including one leveraging streaming APIs for parallel analysis, and show *Ananke* results in small overheads, close to those of existing tools for fine-grained backward provenance.

C1 Introduction

Distributed, large, heterogeneous Cyber-Physical Systems (CPSs) like Smart Grids or Vehicular Networks [91] rely on online analysis applications to monitor device data. In this context, the data streaming paradigm [180] and the DataFlow model [3] enable the inspection of large volumes of continuous data to identify specific patterns [58], [142]. Streaming applications fit CPSs' requirements due to the high-throughput, low-latency, scalable analysis enabled by Stream Processing Engines (SPEs) [2], [8], [10], [29], [152] and their correctness guarantees, which are critical for sensitive analysis. Figure C1a shows two streaming applications, or *queries*, monitoring a vehicular network to spot cars visiting a specific area (Q_1) or speeding (Q_2). Vehicle reports (timestamp, id, position), or *tuples*, arrive every 5 minutes; t^i_j is the i-th tuple from car j, a^i_j the i-th alert from query j. This scenario is our running use-case throughout the paper.

Motivating challenge

CPSs need continuous monitoring for emerging threats or dangerous events [156], [170], which may result in many alerts that analysts are then left to prioritize [56], [173]. For streaming-based analysis, *provenance* techniques [76], [151], which connect results to their contributing data, are a practical way to inspect data dependencies, since breakpoint-based inspection is not fit for live queries that run in a distributed manner and cannot be paused [50]. Existing provenance tools for traditional databases target *backward tracing*, to find which source tuples contribute to a result [20], [38], [47], [50], [76] (Figure C1b),

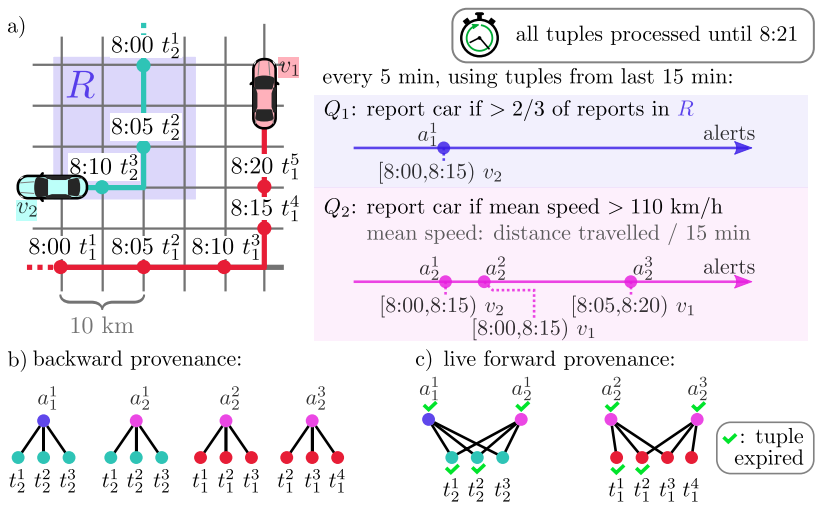


Figure C1: a) Two sample queries to monitor car location and mean speed (all tuples up to time 8:21 are processed), and their b) backward and c) live forward provenance graphs.

and *forward tracing*, to find which results originate from a source tuple [20], [47]; however, streaming-based tools only exist for backward-provenance [76], [151].

The need for live, streaming, forward provenance is multifold: (1) while backward tracing can give assurance on the trustworthiness of end-results [38], forward tracing allows to identify all results linked to specific inputs [47], e.g., to mark all results linked to a privacy-sensitive datapoint (e.g., a picture of a pedestrian, in the context of Vehicular Networks) before such results are analyzed further; (2) live maintenance of the provenance graph avoids data duplication and allows to start the analysis of provenance data safely (e.g., once all sensitive results that could be connected to the aforementioned picture have been marked); (3) a streaming-based forward provenance tool does not require intermediate disk storage (which might be forbidden for pictures taken in public areas) and enables lean dependency and causality analysis of the monitored events. Note that, as shown in our evaluation, providing live, streaming, forward provenance with tools external to the SPE running the monitoring queries incurs significant costs that can be avoided by relying on intra-SPE provenance processing instead.

Contribution

Motivated by the open issues, our key question is:

"Can we enrich data streaming frameworks that deliver backward provenance to efficiently provide live, duplicate-free, fine-grained, forward provenance for arbitrarily complex sets of queries?"

We answer affirmatively with our contributions:

- We formulate the concrete goals and evaluation metrics of solutions for live, duplicate-free, fine-grained, forward provenance.
- We implement a general framework, *Ananke*[(C1)], able to ingest backward provenance and deliver an evolving bipartite graph of live, duplicate-free, fine-grained, forward provenance (or simply live provenance) for arbitrary sets of queries. *Ananke* delivers each result and source tuple contributing to one or more results exactly once, distinguishing source data that could still contribute to more results from *expired* source data that cannot.
- *Ananke*'s key idea builds on our insights on forward provenance w.r.t. the backward provenance problem and defines a simple yet efficient approach, enabling specialized-operator-based implementations, as well as modular ones that utilize native operators of the underlying SPE. We design and prove the correctness of two streaming-based algorithmic implementations: one targeting to optimize the labeling of the expired source data as fast as possible, and one that shows how the general SPEs' parallel APIs are sufficient to parallelize *Ananke*'s algorithm, and thus sustain higher loads of provenance data.

[(C1)] In Greek mythology, Ananke personifies inevitability, compulsion and necessity.

- We conduct a thorough evaluation of our *Ananke* implementation on top of Apache Flink [29], with real-world use cases and data, and also match with previous experiments and an implementation that delivers live forward provenance by relying on tools external to the SPE, for a fair comparison of *Ananke*'s overheads.

The implementations used in our evaluation are open-sourced at [75] for reproducibility. Figure C1c shows *Ananke*'s live provenance assuming both queries have processed all tuples up to time 8:21. Each source and sink tuple appear exactly once in the bipartite graphs. Some tuples are labeled by a green check-mark, indicating that they are expired and will not be connected to future results.

Organization: Section C2 covers preliminary data streaming and provenance concepts. Section C3 provides the definitions we use and also includes a formal problem formulation. Section C4-Section C5 cover our contribution, later evaluated in Section C6. We discuss related work in Section C7 and conclude in Section C8.

C2 Preliminaries

C2.1 Data Streaming Basics

Like Apache Flink [29] (or simply Flink), *Ananke* builds on the DataFlow model [3]. *Streams* are unbounded sequences of *tuples*. Tuples have two *attributes*: the metadata μ and the payload φ, an array of *sub-attributes*. The metadata μ carries the timestamp τ and possibly further sub-attributes. To refer to a sub-attribute of μ , e.g., τ, we use the notation $t.\tau$. We reference φ's i-th sub-attribute as $t.\varphi[i]$ (omitting t when it is clear from the context). In combined notation, a stream tuple is written as $\langle \mu, \phi \rangle = \langle \tau, \ldots, [\varphi[1], \varphi[2], \ldots] \rangle$.

Streaming queries (or simply queries) are composed of *Sources*, *operators* and *Sinks*. A Source forwards a stream of *source tuples* (e.g., events measured by a sensor or reported by other applications). Each *source stream* can be fed to one or more operators, the basic units manipulating tuples. Operators, connected in a Directed Acyclic Graph (DAG), process input tuples and produce output tuples; eventually, *sink tuples* are delivered to *Sinks*, which deliver results to end-users or other applications. In our model, we assume each tuple is immutable. Tuples are created by Sources and operators. The latter can also forward or discard tuples.

As source tuples correspond to events, τ is set by the Source to when that event took place, the *event time*. Operators set τ of each output tuple according to their semantics, while φ is set by user-defined functions. Event time is not continuous but progresses in discrete increments defined by the SPE (e.g., milliseconds). We denote the smallest such increment of an SPE by δ. All major SPEs [7], [8], [10], [29] support user-defined operators but also provide native ones: *Map*, *Filter*, *Aggregate* and *Join*. Since we make use of such native operators, we provide in the following their formal description for self-containment. However, *Ananke* provides live provenance without imposing

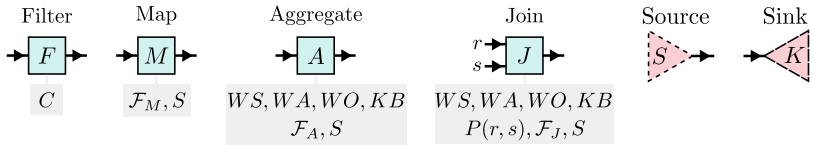


Figure C2: SPEs' native operators, Source, and Sink.

any restriction on the operators of the query. Figure C2 illustrates the native operators, the Source, and the Sink.

We begin with *stateless* operators, which process tuples one-by-one.

A **Filter (F)** relies on a user-defined filtering condition C to either forward an input tuple, when C holds, or discard it otherwise.

A **Map (M)** uses a user-defined function $\mathcal{F}_M$ (to transform an input tuple into $m \geq 1$ output tuples) and S, the schema of the output tuple payloads. It copies the τ of each input into the outputs.

Differently from stateless operators, *stateful* ones run their analysis on *windows*, delimited groups of tuples maintained by the operators. *Time windows* are defined by their size WS (the length of the window), advance WA (the time difference between the left boundaries of consecutive windows), and offset WO (the alignment of windows relative to a reference time; in Flink, this is the Unix epoch). For example, a window having WS, WA, and WO set to 60, 10 and 5 minutes, respectively, will cover periods [00:05,01:05), [00:15,01:15), etc. Consecutive periods covered by a window can overlap when $WA < WS$. Lastly, the left and right boundaries of a window are inclusive and exclusive, respectively. We say a tuple t *falls* in a window [A,B) if $A \leq t.\tau < B$. As windows can overlap, a tuple can fall into one or more windows.

We now present stateful operators in more detail.

An **Aggregate (A)** is defined by: (1) WS, WA, WO: the window size, advance, and offset, (2) KB: an optional key-by function to maintain separate (yet aligned) windows for different key-by values, (3) $\mathcal{F}_A$: a function to aggregate the tuples falling in one window into the φ attribute of the output tuple created for such window, (4) S: the output tuple's payload schema.

When an output tuple is created for a window (and a key-by value, if KB is defined), we assume its timestamp is set to such window's right boundary [7], [10], [29].

A **Join (J)** matches tuples from two input streams, r and s. It keeps two windows, one for r and one for s tuples, which share the same values for parameters WS, WA and WO. Each pair of r and s tuples sharing a common key are matched for every pair of windows covering the same event-time period they fall in. The Join operator relies on the following parameters: (1) WS, WA, WO: the window size, advance, and offset, (2) KB: a key-by function to maintain separate (yet aligned) pairs of windows for different key-by values, (3) $P(t_r, t_s)$: a predicate for pairs of tuples from the two input streams, (4) $\mathcal{F}_J$: a function to create the φ attribute of the output tuple, for each pair of input tuples for which $P(t_r, t_s)$ holds, and (5) S: the schema of the output tuple's payload φ. Similarly to the Aggregate, when an output tuple is created by a

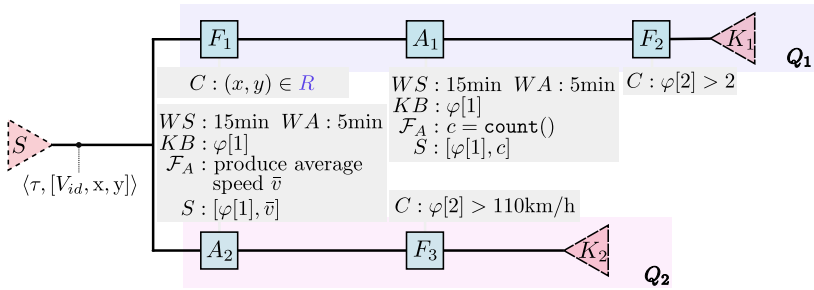


Figure C3: The DAG of Q_1, Q_2 from Figure C1. Source tuples contain reports' timestamp in μ, and the vehicle ID V_{id} and position x, y in φ. Source tuples are forwarded to both queries. Tuples arriving at the Sinks correspond to alerts in Figure C1.

Join, its sub-attribute τ is set to the right boundary of the window.

In the remainder, we (1) differentiate between stateless and stateful only if necessary, we (2) assume $WO = 0$ unless otherwise stated, and (3) assume a stream can be multiplexed to many operators.

Figure C3 presents Figure C1's queries. Source S emits tuples of schema $\langle \tau, [V_{id}, x, y] \rangle$ (timestamp, vehicle ID, x- and y-coordinates). In Q_1, Filter F_1 forwards tuples within region R, Aggregate A_1 counts each car's reports, and Filter F_2 forwards to Sink K_1 only tuples with a count higher than 2 (as tuples arrive every 5 minutes, the count can only be 1, 2 or 3). In Q_2, Aggregate A_2 emits the mean speed of each car within the last 15 minutes. Filter F_3 forwards the tuples whose mean speed[(C2)] exceeds 110km/h to Sink K_2.

C2.2 Watermarks and Correctness Guarantees

Because of asynchronous, parallel, and distributed execution, stateful operators processing tuples from multiple streams can receive such tuples out-of-order. Hence, receiving a tuple with a timestamp greater than some window's right boundary does not imply that tuples received later could not still contribute to said window.

To ensure result correctness for out-of-order streaming processing [99], Aggregate and Join rely on watermarks to make such distinction, as suggested by pioneer as well as state-of-the-art SPEs [29], [99]; the definition is paraphrased here:

Definition C1. *The watermark W_i^ω of operator O_i at a point in wall-clock time[(C3)] ω is the earliest event time a tuple to be processed by O_i can have from time ω on (i.e., $t_i.\tau \geq W_i^\omega, \forall t_i$ processed from ω on).*

[(C2)] In Figure C1, given the grid's cell size, cars' mean speed is 120km/h if covering four cells in three consecutive tuples.

[(C3)] Notice that from here on, we only differentiate between wall-clock time (or simply time) and event time if such distinction is not clear from the context.

Watermarks are created periodically by Sources and propagate as special tuples through the DAG[(C4)]. Upon receiving a watermark, an operator stores the watermark's time, updates its watermark to the minimum of the latest watermarks received from each of its input streams and forwards its watermark downstream. For an Aggregate or Join O_i, every time the watermark advances from W_i^{ω} to $W_i^{\omega'}$, an output tuple is created for each window maintained by O_i that has a right boundary less or equal to $W_i^{\omega'}$. If multiple results are created, they are emitted in event-time order.

C2.3 Backward Provenance

Ananke aims at extending frameworks that provide backward provenance (Section C1). Such frameworks [76], [77], [151] rely on *instrumented* operators, i.e., wrappers that add extra functionality to operators. Our contribution can extend any streaming framework providing backward provenance, assuming each sink tuple has additional sub-attributes in μ that can be used to retrieve the unique source tuples contributing to it. Such sub-attributes can be pointers to source tuples [151] or IDs identifying source tuples maintained in a dedicated provenance buffer [76], [77]. In the remainder, we rely on a general function `get_provenance` to retrieve backward provenance.

C3 Definitions and Problem Statement

This section includes the definitions we use to present and prove our contribution's correctness, and a formal statement of our goals.

Definition C1. *We say a tuple* t contributes directly *to another tuple* t^* *if an operator produces* t^* *based on the processing of* t *and write:* $t \rightarrow t^*$. *We then say* t contributes *to* t^* *and use the notation* $t \rightsquigarrow t^*$ *if* $t \rightarrow t' \rightarrow t'' \rightarrow \ldots \rightarrow t^*$. *Thus, if* $t \rightarrow t^*$, *then* $t \rightsquigarrow t^*$.

From the above, a source tuple t_S from Source S *contributes* to a sink tuple t_K received by Sink K, if there is a directed, topologically-sorted path $S, O_1, \ldots, O_i, \ldots, O_k, K$ and a sequence of tuples $t_S = t_0, \ldots, t_{i-1}, t_i, \ldots, t_k = t_K$, s.t. $\forall i = 1, \ldots, k$, where t_{i-1} and t_i are input and output tuples of O_i, and $t_{i-1} \rightarrow t_i$ for all $i = 1, \ldots, k$.

Definition C2. *At time* ω, *tuple* t *is* active *if it can still contribute directly to a tuple produced by an operator. Otherwise,* t *is* inactive. *At time* ω, *tuple* t *is* alive *if it is* active *or if there is at least one active tuple* t^* *such that* $t \rightsquigarrow t^*$. *Otherwise,* t *is* expired.

For instance, if a Map produces a tuple t_2 upon ingesting tuple t_1, the latter is *inactive*, but remains *alive* as long as t_2 is being processed downstream (or as long as t_2 itself is alive). Note that all sink tuples are expired by definition. Using the above, we define the live provenance to be delivered by *Ananke*:

[(C4)] Notice that this assumption about *in-band* watermarks is not a constraint. Different watermarking schemes that could also be adopted are discussed in e.g., [99], [125].

Definition C3. *At time ω in the execution of a set of queries $\mathbb{Q}$, being $\mathbb{K}_\mathbb{Q}$ the set of $\mathbb{Q}$'s Sinks, the live, duplicate-free, bipartite graph forward provenance $\mathbb{F}(\mathbb{Q}, \omega)$ consists of (1) a set of vertices V, containing exactly one vertex for each sink tuple forwarded to $\mathbb{K}_\mathbb{Q}$, and exactly one vertex for each source tuple contributing to any sink tuple forwarded to $\mathbb{K}_\mathbb{Q}$, (2) a set of edges E, each edge connecting a sink tuple with its contributing source tuples, and (3) a set of "expired" labels L, one for each vertex in V if the tuple it refers to is expired.*

Notice that if, at time ω, a vertex in V is alive, then more edges connecting such vertex to other vertices could be found later in the execution of $\mathbb{Q}$. In Figure C1c, which shows $\mathbb{F}(\mathbb{Q}, 8{:}21)$, new edges could connect the vertices of tuples not marked as expired to vertices referring to sink tuples that have not yet been produced.

Given the preceding definitions, we now formulate our goals and the necessary requirements to reach these (using prefix R- for the latter). For brevity, we refer to vertices containing a source tuple or a sink tuple as *source vertices* and *sink vertices*, respectively, and use the expression *expired* for tuples and vertices interchangeably.

Problem formulation Being $\mathbb{B}_\mathbb{Q}$ a set of streams delivering $\mathbb{K}_\mathbb{Q}$'s sink tuples with backward provenance retrievable via attribute μ, the goal is to continuously deliver $\mathbb{F}(\mathbb{Q}, \omega)$'s V, E and L, for increasing values of ω, as one or multiple streams, to meet the following requirements:
(R-V) Each vertex referring to a source or a sink tuple is delivered exactly once, by a tuple $\langle \tau_V, [ID_S, t_S] \rangle$ or $\langle \tau_V, [ID_K, t_K] \rangle$, respectively. ID_i is a unique ID for the vertex associated to t_i;
(R-E) each edge between vertices ID_S and ID_K is delivered exactly once by a tuple $\langle \tau_E, [ID_S, ID_K] \rangle$, with τ_E greater than or equal to the timestamp τ_V of the connected vertices; and
(R-L) an "expired" label is delivered once for each vertex ID_i by a tuple $\langle \tau_L, [ID_i] \rangle$, with $\tau_L \geq \tau_E$, for each edge E adjacent to ID_i.

For the queries in Figure C1a, Figure C4 shows $\mathbb{F}(\mathbb{Q}, 8{:}16)$ and the tuples delivered to update it to $\mathbb{F}(\mathbb{Q}, 8{:}22)$.

While referring to a set of queries $\mathbb{Q}$ and a set of Sinks $\mathbb{K}_\mathbb{Q}$ for generality, our problem formulation is justified even for a single query with exactly one Source and Sink, because subsequent sink tuples can have overlapping sets of

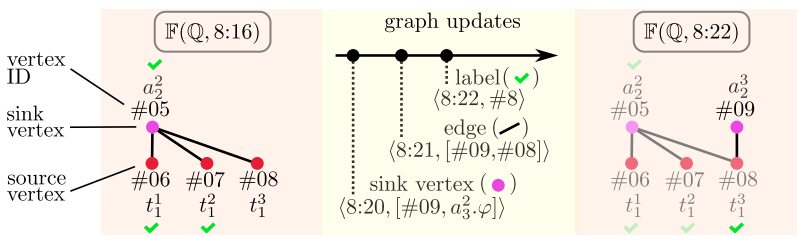


Figure C4: Live provenance graph for Figure C1's queries at 8:16 and 8:22, and the stream of graph tuples received in between.

source tuples (as in the example of Figure C1), and each such source tuple still needs to be delivered exactly once and later marked as expired.

Performance metrics For provenance to be practical in real-world applications, its performance overheads need to be small. The efficiency of our solution is evaluated through its overhead on the following metrics:

- *Throughput*, number of tuples a query ingests per unit of time.
- *Processing Latency*, the delay in the production of a sink tuple after all its contributing source tuples have arrived at the query.
- *CPU utilization*, the percentage of the total CPU time a query utilizes, across all available processors (0-100%).
- *Memory consumption*, the amount of RAM a query utilizes.

We also introduce the *provenance latency* metric, to quantify the event time it takes for $\mathbb{F}(\mathbb{Q}, \omega)$'s components to become available:

- For tuple t's vertex V, it is computed as $\tau_V - t.\tau$.
- For an edge E, it is computed as $\tau_E - max(\tau_V^S, \tau_V^K)$, where τ_V^S and τ_V^K are the timestamps of the vertices connected by the edge.
- For the label L of some vertex, it is computed as $\tau_L - \tau_V$.

Implementation requirements We want *Ananke* to be a streaming-based extension (of $\mathbb{Q}$) that delivers the vertices, edges, and labels of $\mathbb{F}(\mathbb{Q}, \omega)$ through its output stream(s). A solution can rely on user-defined or native operators (Section C2). Being a streaming-based extension of $\mathbb{Q}$, the latter's watermarks are propagated to *Ananke* operators, too. For generality, we assume a user-defined operator needs to support two methods (not invoked concurrently by the SPE): `on_tuple(`t`)`, invoked upon reception of tuple t, and `on_watermark(`W`)`, invoked when the watermark W is updated.

C4 Discerning Alive and Expired Tuples

Live provenance needs to discern alive from expired tuples. Even if sink tuples cannot contribute directly to other tuples, source tuples can be inactive but alive. As we show, alive and expired tuples can be separated using static query attributes and the Sink watermarks.

Figure C5 illustrates how a source tuple's contribution "ripples" through event time for a query composed of Aggregates A_1 and A_2, Map M, and Filter F. A source tuple, t_S, falls into two windows of A_1, which emit two outputs that pass through M and contribute to three separate windows in A_2. Two of the three output tuples of A_2 get dropped by F, and a single tuple t_K arrives at K. When that happens, it is unknown whether t_S will contribute to more sink tuples - for example, it is uncertain if F will drop the next output of A_2. The

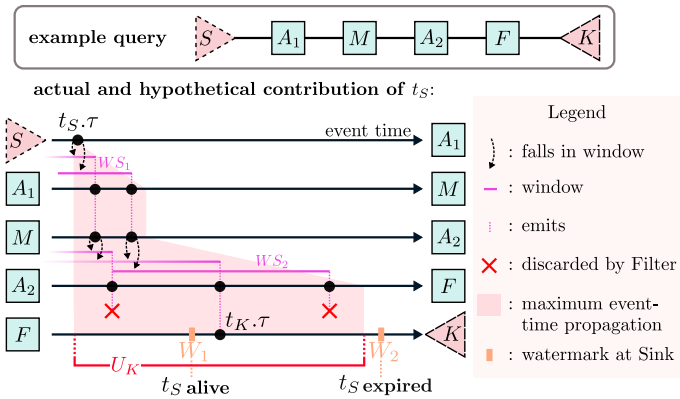


Figure C5: Sample query showing actual and maximal contributions of a source tuple to downstream tuples.

figure also explores t_S's hypothetical *maximal* contributions: We ignore exact window placements and examine the extreme case, tuples always falling at the beginning or the end of windows. We shade all event times that can contain (direct or indirect) contributions of t_S. As shown, the growth of the shaded region only depends on the window size of stateful operators. Tuple t_S can contribute to any tuple inside the shaded region. Thus, if K's watermark falls after that region (W_2 in the example), then t_S is surely expired. U_K denotes the width of the shaded region. With this intuition, we proceed with proving the following theorem:

Theorem C1. *Given $\mathbb{K}_\mathbb{Q}$ (Definition C3), it is possible to statically compute constants U_K, one for each Sink $K \in \mathbb{K}_\mathbb{Q}$ so that: If a source tuple t_S has contributed to a sink tuple t_K that arrives at K at time ω or later, then:*

$$t_S.\tau \geq W_K^\omega - U_K. \tag{3.1}$$

Inversely, if $t_S.\tau < \min_K(W_K^\omega - U_K)$, then t_S is expired and cannot contribute to any *sink tuple fed to $\mathbb{K}_\mathbb{Q}$ at time ω or later.*

We move bottom-up towards the proof. First, we study pairs of chained operators, each with one downstream peer in Lemma C1. In Lemma C2, we focus on longer operator chains before examining arbitrary paths between sets of operators in Corollary C1 and finally concluding with the proof of Theorem C1. Here, we use the term *operator* in a broad sense, also to refer to Sources and Sinks.

Lemma C1. *For any operator O_i with downstream operator O_{i+1}, and a tuple t_{i+1} arriving at O_{i+1} at time ω or later, it holds that if an input tuple t_i of O_i contributed to t_{i+1}, then: $t_i.\tau \geq W_{i+1}^\omega - WS_i$.*

Proof of Lemma C1. From the way stateful operators set their output timestamps (Section C2), we obtain $t_i.\tau \geq t_{i+1}.\tau - WS_i$ [(C5)]. Furthermore, from

[(C5)] Although the assumption about how timestamps are set covers commonly used SPEs, the

Definition C1 we get $t_{i+1}.\tau \geq W^{\omega}_{i+1}$. Thus, Lemma C1 follows immediately and it also holds for a stateless O_i by setting $WS_i = 0$.

□

We now expand the proof to chains of operators.

Lemma C2. *For any chain of operators $O_1 \dots O_n$, their downstream operator O_{n+1} and a tuple t_{n+1} that arrives at O_{n+1} at time ω or later, it holds that if an input tuple t_1 of operator O_1 contributed to t_{n+1}, then $t_1.\tau \geq W^{\omega}_{n+1} - \sum_{j=1}^{n} WS_j$.*

Proof of Lemma C2. We begin by showing that:

$$\forall t_1, t_{n+1}, \quad t_1 \rightsquigarrow t_{n+1} \Rightarrow t_1.\tau \geq t_{n+1}.\tau - \sum_{j=1}^{n} WS_j. \tag{3.2}$$

Let us denote as t_i tuples arriving at operator O_i, with $t_i \rightarrow t_{i+1}$ for $i \in [1, n+1]$. From the proof of Lemma C1, we know that $t_1.\tau \geq t_2.\tau - WS_1$. Plugging in this relation again into the first term on the right-hand side of the inequality, we obtain $t_1.\tau \geq t_2.\tau - WS_1 \geq t_3.\tau - WS_2 - WS_1$. Performing this step n times yields Equation 3.2. Given Definition C1, $t_{n+1}.\tau \geq W^{\omega}_{n+1}$; hence:

$$\forall t_1, t_{n+1},\ t_1 \rightsquigarrow t_{n+1} \Rightarrow t_1.\tau \geq t_{n+1}.\tau - \sum_{i=1}^{n} WS_i \geq W^{\omega}_{n+1} - \sum_{i=1}^{n} WS_i.$$

□

Corollary C1. *Given a set of operators $\mathbb{O} = \{O_i\}$ connected to operator O_X through a set of paths $\mathbb{P}$, for any tuple t_X fed to O_X at time ω or later, it holds that if an input tuple t_i of O_i contributed to t_X, then $t_i.\tau \geq W^{\omega}_{out} - \max_{p \in \mathbb{P}} S_p$ where $S_p = \sum_{j \in p} WS_j$.*

The above corollary follows directly from Lemma C2 and allows us to compute the maximum "delay" between tuples traversing the longest path in a query, enabling us to prove Theorem C1.

Proof of Theorem C1. To prove Equation 3.1, we apply Corollary C1 to any sink tuple t_K arriving at Sink K at time ω or later, and any source tuple t_S, which gives $t_S.\tau \geq W^{\omega}_{K} - U_K$. The constants $U_K = \max_{p \in \mathbb{P}} S_p$, with $\mathbb{P}$ the set of all paths to sink K, can be computed statically based on the attributes of the query graph. □

The following remark, stemming directly from Theorem C1, introduces a per-application safety-margin for expired tuples.

analysis holds also for any output timestamp within the window boundaries. If the output tuples of stateful operator O_i have timestamps that are Δ_i from the left boundary L_i of the window, it holds that $t_{i+1}.\tau \leq L_i + \Delta_i$. By definition of the left boundary, for any tuple t_i in the window it is true that $t_i.\tau \geq L_i$ and the relation becomes $t_i.\tau \geq t_{i+1}.\tau - \Delta_i$. Since WS_i is the maximum value of Δ_i, using WS_i will always give correct results (possibly with a higher delay).

Remark C1. *If we define* $U = \max_K U_K$*, then any source tuple* t_S *with* $t_S.\tau < \min_K W_K^\omega - U$ *is expired.*

C5 Algorithmic Implementation

As mentioned in Section C2, SPEs provide native operators and support user-defined ones. Here we show how *Ananke*'s goals can be met by a user-defined operator (ANK-1, Section C5.1) or by composing native operators (ANK-N, Section C5.2). While ANK-1 targets the prompt final labeling of $\mathbb{F}(\mathbb{Q}, \omega)$'s vertices, ANK-N shows how the APIs for parallel execution commonly provided by SPEs are sufficient to parallelize *Ananke*'s algorithm. We study their trade-offs in Section C6.

Both in ANK-1 and ANK-N, the ID of t_S's source vertex is based on t_S's attributes. Hence, source tuples with equal attributes refer to the same source vertex. As each sink tuple represents a unique event, it results in a sink vertex with a unique ID. Since each sink tuple can carry each source tuple at most once in its provenance, edges are also unique. We discuss in Section C5.3 how to extend *Ananke* to other ID policies. In the following, we make use of Remark C1 for distinguishing alive from expired tuples. As mentioned in Section C3, the set of streams $\mathbb{B}_\mathbb{Q}$, delivering backward provenance to *Ananke*, forwards the required watermarks. For both implementations, we show how they meet the requirements for vertices (R-V), edges (R-E), and labels (R-L) from Section C3.

C5.1 ANK-1: Single User-defined Operator

As introduced in Section C3, user-defined operators must support two methods: `on_tuple` and `on_watermark`. Algorithm C1 covers such methods for ANK-1; methods `unique_id()` and `get_id(`t`)` respectively generate a unique ID and compute the ID of t based on its attributes.

Claim C1. *A user-defined operator fed* $\mathbb{B}_\mathbb{Q}$ *can correctly deliver* $\mathbb{F}(\mathbb{Q}, \omega)$ *with Algorithm C1's* `on_tuple` *and* `on_watermark` *methods.*

Proof. Upon reception of sink tuple t_K from $\mathbb{B}_\mathbb{Q}$, ANK-1 emits exactly once the corresponding (1) sink and (2) source vertex, only if its ID was not stored in the timestamp-sorted set T (i.e., if the corresponding source vertex was not forwarded before), thus meeting requirement (R-V); (3) edges, and (4) sink vertex label (L1-11). Set T represents ANK-1's "memory" about forwarded source vertices. The emitted tuples carry as timestamp the current watermark value, thus meeting requirements (R-E), as the edge does not precede the vertices, and (R-L) for the sink vertices.

Each source vertex ID_S in T is purged once the corresponding source tuple is expired, i.e., when $W - U$ is greater than its τ_V (L12-17). Upon purging of ID_S, exactly one "expired" label is generated for the corresponding source vertex, with the current watermark as the timestamp. Since watermarks are strictly increasing, it is guaranteed that each label has a timestamp higher than or equal to that of its source vertex, meeting (R-L) for the source vertex. □

Algorithm C1 ANK-1 algorithmic implementation

```
    data Set T of pairs (τ, ID), ordered on τ, and watermark W
 1: procedure ON_TUPLE(t_K)
 2:     ID_K = UNIQUE_ID()
 3:     EMIT(⟨W, [ID_K, t_K]⟩)                          ▷ Emit sink vertex
 4:     sourceTuples ← GET_PROVENANCE(t_K)
 5:     for t : sourceTuples do
 6:         ID_S = GET_ID(t_S)
 7:         if (t.τ, ID_S) ∉ T then
 8:             EMIT(⟨W, [ID_S, t_S]⟩)                  ▷ Emit source vertex
 9:             T ← T ∪ {(t.τ, ID_S)}
10:         EMIT(⟨W, [ID_S, ID_K]⟩)                     ▷ Emit edge
11:     EMIT(⟨W, [ID_K]⟩)                               ▷ Emit sink vertex label

12: procedure ON_WATERMARK(W)
13:     for (τ, ID_S) ∈ T do
14:         if τ >= W − U then
15:             output
16:         EMIT(⟨W, [ID_S]⟩)                           ▷ Emit source vertex label
17:         T ← T \ (τ, ID_S)
```

C5.2 ANK-N: Native Operator Composition

We now present ANK-N, based on native operators. First, we study the case of $U > 0$. For ease of exposition, we initially rely on two auxiliary stateful operators, *Delay* (D) and *Forward Once* (FO), that help meet the requirements, and later show how D's and FO's semantics can be satisfied by native operators. Finally, we cover the case $U = 0$, where all $\mathbb{Q}$'s operators are stateless.

A **Delay (D)** operator produces, for each input tuple t with a unique payload φ, an output tuple t' as a copy of t with $t'.\tau = \texttt{delay}(t.\tau) := \left(\lfloor \frac{t.\tau}{U} \rfloor + 2\right) \cdot U$, $t'.\varphi = t.\varphi$ and $U < t'.\tau - t.\tau \leq 2U$.

A **Forward Once (FO)** guarantees that, whether one or more tuples are fed to it sharing the same ID sub-attribute, only the earliest such tuple is output, with its payload unchanged but its timestamp delayed by `delay()` as in D. After such tuple is output, FO produces nothing for the subsequent input tuples with that ID until a period of U has passed in the input stream. As identical source tuples that appear at different times in $\mathbb{B}_{\mathbb{Q}}$ are not spaced apart further than U (Remark C1) and have the same ID, FO will output unique source tuples exactly once.

ANK-N overview: Using D, FO and native operators, we construct the DAG of Figure C6 to meet the requirements from Section C3, with $\mathbb{B}_{\mathbb{Q}}$ as input. First, we outline the main idea. Let us consider sink tuple t_K from $\mathbb{B}_{\mathbb{Q}}$, and follow its path through the DAG.

Upon processing t_K, M_u produces a sink vertex that carries a copy of t_K, a unique ID_K and the character "K" as sub-attributes of its payload. M_u

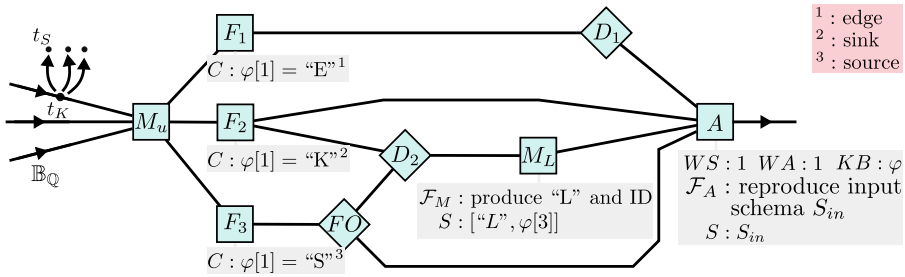


Figure C6: Overview of ANK-N. Algorithm C2 shows $\mathcal{F}_{M_u}$.

also produces, for all source tuples in t_K's provenance, a source vertex with character "S" (carrying an ID based on the source tuple attributes) as well as an edge. Each edge carries the character "E" and the IDs of the source and sink tuples that it connects. In the proof of the following claim, we continue tracing the paths of "K", "E" and "S" tuples and show that all requirements for delivering a live provenance graph are met.

Claim C2. *The DAG in Figure C6, using the D and FO operator, as well as native ones with the mapping function M_u defined in Algorithm C2, once fed $\mathbb{B}_\mathbb{Q}$, correctly delivers $\mathbb{F}(\mathbb{Q}, \omega)$.*

Proof. We first prove that for each source tuple t_S, its vertex, edges, and label are delivered correctly:
(1) *Ensuring t_S's vertex is created once.* t_S can appear multiple times, as provenance of multiple sink tuples. Based on Theorem C1, after contributing to sink tuple t_K (timestamped τ_K), t_S cannot contribute to later sink tuples timestamped $\geq \tau_K + U$. Thus, no pair of source tuples with the same ID can be farther away than U.

As source vertices (marked with "S") are forwarded to FO, which is defined to output each source vertex with a given ID exactly once with $\tau_S = \texttt{delay}(\tau_K)$, (R-V) is met for source vertices.
(2) *Ordering t_S's edges behind the vertex.* For every t_S in the provenance of t_K, M_u produces the connecting edge "E". For these edges to come after t_S's vertex, they are forwarded by F_1 to D_1, which outputs copies of each edge, with $\tau_E = \texttt{delay}(\tau_K)$, meeting (R-E).

Algorithm C2 Map function $\mathcal{F}_{M_u}$ of M_u

```
1: function OUT=F_{M_u}(t_K)
2:     ID_K = UNIQUE_ID()
3:     out.ADD(["K", t_K, ID_K])                ▷ Add sink vertex
4:     for t_S in GET_PROVENANCE(t_K) do
5:         ID_S = GET_ID(t_S)
6:         out.ADD(["S", t_S, ID_S])            ▷ Add source vertex
7:         out.ADD(["E", (ID_K, ID_S)])         ▷ Add edge
8:     output out                               ▷ Produce tuples
```

(3) *Producing t_S's label correctly.* The latest edge E' involving t_S could be produced by M_u at event time $\tau_K + U - \epsilon$ (for $\epsilon > 0$), according to Remark C1. As E' will be delayed to $\tau'_E = \texttt{delay}(\tau_K + U - \epsilon)$ the source label tuple must be delayed beyond τ'_E to meet (R-L). From FO, the source vertex (which has been delayed already) is multiplexed to D_2, delayed again, and mapped by M_L to a label tuple with timestamp $\tau_{S,L} = \texttt{delay(delay}(\tau_K)) = \texttt{delay}(\tau_K) + 2U$ - which is strictly greater than τ'_E, meeting (R-L) for source tuples.

Thus, the components involving t_S are meeting the requirements.

We now focus on sink tuple t_K's vertices, edges, and labels:

(1) *Ensuring t_K's vertex is created once.* F_2 forwards the single instance of t_K's vertex (timestamp τ_K), meeting (R-V) for sink tuples.

(2) *Ordering t_K's edges behind the vertex.* As explained in (2) above, edges involving t_K are delayed to $\tau_E = \texttt{delay}(\tau_K)$ and (R-E) is met.

(3) *Producing t_K's label correctly.* The label for t_K's vertex must not have a lower timestamp than any edge connected to t_K's vertex, and these edges are delayed. As the sink vertex "K" is multiplexed from F_2 to D_2 and mapped to a label by M_L, the resulting label has timestamp $\tau_{K,L} = \texttt{delay}(\tau_K) = \tau_E$, meeting (R-L) for sink tuples.

Thus, the components involving t_K also meet the requirements.

Lastly, we now show how Aggregate A emits vertices, edges, and labels in order. Each tuple, timestamped τ, falls in one window $[\tau, \tau + \delta)$ of A, and a copy of each tuple is produced by A when A receives a watermark $> \tau + \delta$. As Aggregates emit results in timestamp-order (Section C2), this effectively sorts A's outputs. Thus, ANK-N correctly delivers live provenance, meeting the requirements in Section C3. □

We now construct D and FO using native operators:

Delay An Aggregate A, Filter F and Map M (as in Figure C7) can enforce this operator's semantics. A and F create the delay, while M restores the input tuple's payload, creating a delayed copy of it. As A's window size is twice as big as its window advance and $KB : \varphi$, any input tuple with a unique payload falls into two windows and contributes directly to two output tuples of A, with timestamps spaced U apart. As each output tuple t^o of A carries the timestamp of its corresponding input tuple (τ_{orig}), the delay induced on the input tuples can be computed as $t^o.\varphi[1] - t^o.\tau$. F ensures that this delay is greater than U, which is always the case for exactly one of the two tuples produced by A for each input tuple - the other is delayed at most U and thus

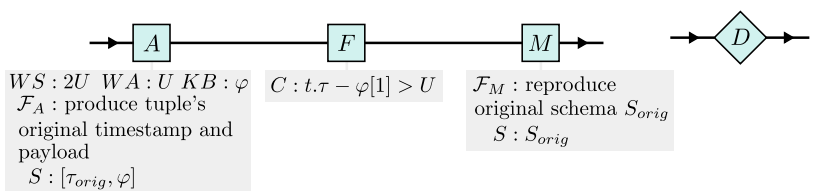


Figure C7: Composition of the D operator.

discarded. In the extreme case, an input tuple t can be delayed by A by $2U$ (the window size), namely if $t.\tau$ coincides with a window's left boundary. The window advance dictates that output tuples produced from t have timestamps spaced U apart. Thus, there will also be a tuple delayed by U produced by A - however, this tuple will be discarded by F. This earlier output tuple, in all other cases where t does not coincide with a window's left boundary, will be delayed even less, and thus also discarded. From this discussion, it is also apparent that the delay for two input tuples $t_1, t_2 | t_1.\tau = t_2.\tau$ is identical (as both t_1 and t_2 are equally distanced from the left boundaries of the windows they fall in).

Forward Once FO ensures that from a group of tuples $t_1, \ldots, t_n$ with increasing timestamps, common key K and $t_n.\tau - t_1.\tau < U$, exactly one tuple t_{FO} with payload $t_n.\varphi$ is produced. This tuple is delayed from t_1 by at least U and at most $2U$. Figure C8 shows how FO can be constructed using two Aggregates A_1 and A_2 and a Join J, to satisfy the required semantics. J's predicate is defined as:

$$\texttt{FOpredicate(r,s)} = ((r.\tau \leq s.\tau) \wedge (r.\varphi[2] \leq s.\varphi[2])) \vee ((s.\tau \leq r.\tau) \wedge (s.\varphi[2] \leq r.\varphi[2])), \tag{3.3}$$

where $\varphi[2]$ is the count emitted by the Aggregate operators.

The group of input tuples to FO are multiplexed to both A_1 and A_2. Since $t_n.\tau - t_1.\tau < U$, and the windows of A_2 are offset by U, $t_1, \ldots, t_n$ will land in either (1) two windows of one Aggregate and one window of the other, or (2) in exactly one window in both Aggregates.

Figure C9 exemplifies how FO achieves the required exactly-once forwarding for both cases [(C6)]. A_1 and A_2 produce output tuples that carry the common payload of the input tuples and the count c of input tuples per window. Their window alignment guarantees that a tuple produced by A_1 or A_2 lands in exactly two of J's windows. Let us now explain the two different cases in detail:

- if $t_1, \ldots, t_n$ fall in one A_1 and A_2 window, output tuples r and s are then fed to J. Since $r.\tau < s.\tau$ and $r.\varphi[2] = s.\varphi[2]$, predicate 3.3 holds. When J emits t_{FO}, it holds that $t_1.\tau + U < t_{FO}.\tau \leq t_1.\tau + 2U$.

[(C6)] The case in which $t_1, \ldots, t_n$ fall in one window for both A_1 and A_2 but A_1's window ends later than A_2's one, and the case in which $t'_1, \ldots, t'_n$ fall in two A_2 windows and one A_1 window are given by "swapping" A_1 and A_2.

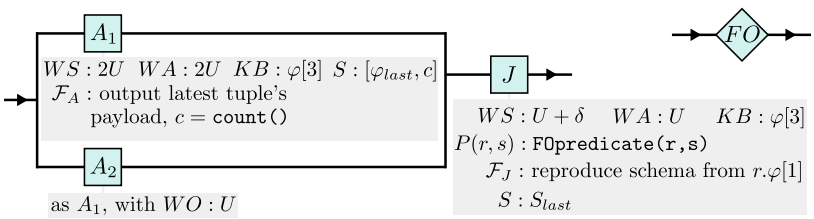


Figure C8: Composition of the FO operator.

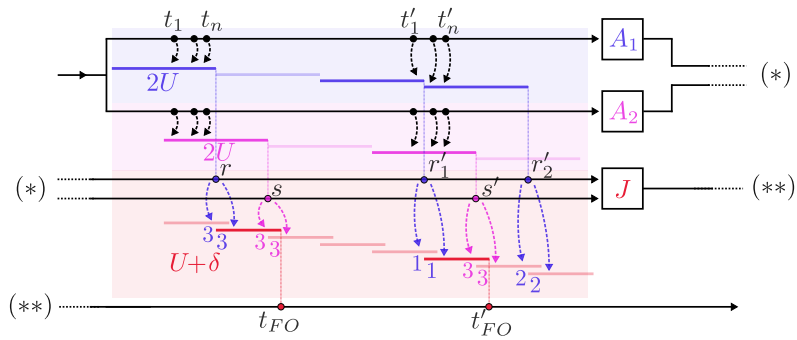


Figure C9: Illustration of how two different groups of input tuples (non-primed and primed) are processed by FO's operators. Each group leads to the emission of one tuple.

- if $t'_1, \ldots, t'_n$ fall in two A_1 windows and one A_2 window, output tuples r'_1, r'_2, s' are fed to J. Then, two windows of J have one tuple each from A_1 and A_2; however, predicate 3.3 holds only for the earlier of the two windows, in which r has lower timestamp and lower count c. Also in this case, $t'_1.\tau + U < t'_{FO}.\tau \leq t'_1.\tau + 2U$.

Thus, in both cases, the input is deduplicated and delayed, and the timestamp of the output tuple t_{FO}/t'_{FO} is given by `delay`(t_1/t'_1).

Corner case

If all operators in $\mathbb{Q}$ are stateless, then $U = 0$. This corner case is not covered by the above implementation of ANK-N, as D and FO cannot have $WS = 0$. If $U = 0$, each sink tuple and its single provenance source tuple have the same timestamp; thus, source tuples are immediately expired once event time passes beyond their timestamp. Each sink tuple will be contributed to by a single source tuple; however, the latter could contribute to several sink tuples. One approach for $U = 0$ is to replace D and FO with identity Maps. The final Aggregate A will then deduplicate source vertices (and source labels, which could now be duplicated as well), as they share the same payload and fall into the same window.

C5.3 Extensions

Ananke associates each sink tuple with a dedicated sink vertex. Hence, our implementations do not need to deduplicate sink vertices, edges, or sink vertex labels. Modifying *Ananke* to allow distinct sink tuples to refer to the same sink vertex and perform that deduplication is nonetheless trivial, as it simply requires to store the sink vertex IDs which have already been forwarded (ANK-1) or to replace the D operators of Figure C6 with FO operators (ANK-N).

C6 Evaluation

We study *Ananke*'s performance relative to the state-of-the-art framework GeneaLog [151]. In Section C6.1, we compare the performance of queries (1) without provenance, (2) with GeneaLog's backward provenance, and (3) with *Ananke*'s live, forward provenance (ANK-1 and ANK-N, cf. Section C5). In Section C6.2, we evaluate the provenance latency for the same use-cases. In Section C6.3, we compare ANK-1 and ANK-N in-depth, studying their performance for various configurations. Finally, in Section C6.4 we highlight *Ananke*'s strengths in comparison to ad-hoc implementations relying on tools external to the SPE.

Ananke Implementation

Ananke is implemented in Java in Flink [29]. It instruments the queries without altering the SPE and uses GeneaLog for backward provenance. We extended GeneaLog to handle tuples arriving at windows of stateful operators out of timestamp order (e.g., when there is parallelism). Moreover, GeneaLog requires operator and tuple objects to *inherit* provenance-specific code. This *non-transparent* (or optimized) implementation can introduce a non-negligible development and maintenance overhead, as implementations need to be altered tying the query implementation to the provenance framework. Here we introduce an alternative *transparent* implementation (denoted by suffix /T), which is based on *encapsulation*: The query is decoupled from the provenance capture, which can be enabled through an automated process. As illustrated in Listing 3.1, the developer simply encapsulates each Flink operator function with the appropriate framework decorators. Those decorators encapsulate the tuples inside special meta-tuples that contain the provenance metadata populated according to the semantics of the underlying operator function, constructing the provenance graph without user intervention. While more flexible, the extra abstraction layers of this technique can increase the data serialization overhead and lower performance. We study both techniques and let the user choose between flexibility and performance.

Evaluation Setup

To account for the broad range of modern CPSs' devices, we use (1) Odroid-XU4 [88] devices (or simply *Odroid*), mounting Samsung Exynos5422 Cortex-

```
1 // Provenance decorators highlighted in blue
2 ANK.source(sourceStream)
3 .filter(ANK.filter(t -> t.type()==0 && t.speed()==0))
4 .keyBy(ANK.key(t -> t.getKey()))
5 .window(SlidingEventTimeWindows.of(WS, WA))
6 .aggregate(ANK.aggregate(new AverageAggregate()))
```

Listing 3.1 Transparently instrumenting a query

A15 2Ghz and Cortex-A7 Octa core CPUs, 2 GB RAM, running Ubuntu 18.04.2, OpenJDK 1.8.0_252, and Flink 1.10.0 (pinned to the four big cores); and (2) a single-socket Intel Xeon-Phi server with 72 1.5GHz cores with 4-way hyper-threading, 32KB L1 and 1MB L2 caches, 102 GB RAM, running CentOS 7.4, OpenJDK 1.8.0_161, and Flink 1.10.0. The execution environment is made explicit in each experiment.

We study the average *throughput*, *latency*, *CPU* and *memory* utilization (Section C3). For real-world use-cases (Section C6.1), we also study the provenance latency. These experiments are repeated at least ten times and are at least ten minutes long. Results are presented as averages with 95% confidence intervals between repetitions. Unless otherwise stated, the parallelism of all operators is set to one. We evaluate the scalability of ANK-N separately in Section C6.3.

C6.1 Comparison with the State-of-the-art

To compare with GeneaLog, we study four queries from the domain of CPSs [151], targeting smart highways and smart grids, and four from smart vehicular systems. The latter are real-world examples from the automotive industry, with broader provenance characteristics, more operators, and larger data volumes. To show *Ananke*'s support for multiple Sinks, we run queries from the same domain together and present the aggregated performance results. Each experiment explores all configurations in Table C1.

Table C1: Query configurations explored in the evaluation.

	NP	GL	ANK-1	ANK-1/T	ANK-N	ANK-N/T
Provenance	-	Backward	Live	Live	Live	Live
Native Ops	-	No	No	No	Yes	Yes
Transparent	-	No	No	Yes	No	Yes

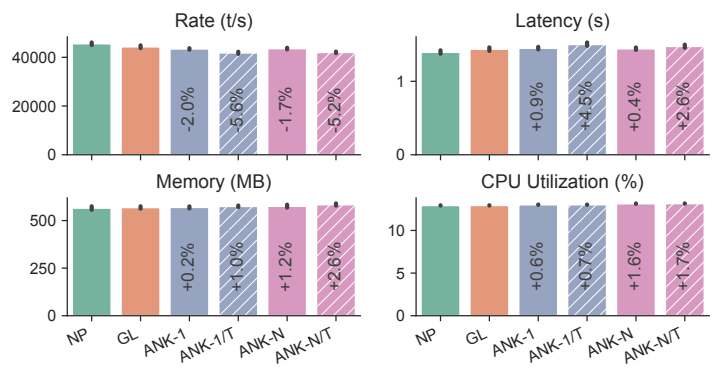


Figure C10: Performance - Linear Road queries.

Linear Road We run two queries from the Linear Road Benchmark [12] on an Odroid. The first query detects broken-down vehicles through consecutive reports of zero speed and constant position. The second detects accidents from cars stopped at the same position. Both queries receive reports from vehicles every 30 seconds, and contain Aggregates and Filters (we refer the reader to [151] for more details). Each sink tuple depends on 4 source tuples in the first query and 8 in the second. Figure C10 shows the query performance without provenance (NP), with GeneaLog (GL), and *Ananke* with the user-defined operator (ANK-1, ANK-1/T) or the native operators (ANK-N, ANK-N/T). The text in *Ananke*'s bars shows the percentage difference from GL. The performance impact of both GL and *Ananke* is small. GL results in about a 3% performance drop for rate and latency and *Ananke* causes a further drop of 2% for ANK-1 and ANK-N and up to 5.6% for the transparent variants (/T).

Smart Grid Figure C11 shows the performance of two queries from the smart grid domain, run on an Odroid. The first reports long-term blackouts by identifying meters with zero consumption for 24 hours. The second detects anomalies, through meters that report abnormal consumption at midnight as compensation for the previous day. Both queries receive hourly power measurements and use Aggregates and Filters; the second also has a Join (we refer the reader to [151] for more details). On average, a sink tuple depends on 192 source tuples in the first query and 24 in the second. The provenance overhead is higher here since the queries have larger aggregation windows and higher volumes of provenance data. GL results in a 9% rate drop and 3% latency increase, while ANK-1 (/T) causes a further 2.7% (14.1%) drop in the rate and a jump of 2.6% (5.2%) in the CPU. For ANK-N (/T), the rate drops by 7.2% (16.8%) compared to GL and the latency rises by at most 2.5%, ANK-N's higher number of operators causes a jump in the CPU, around 14%.

Vehicle Tracking queries This use-case is based on Figure C1. It uses the GeoLife dataset, composed of 18670 GPS traces of various vehicles over

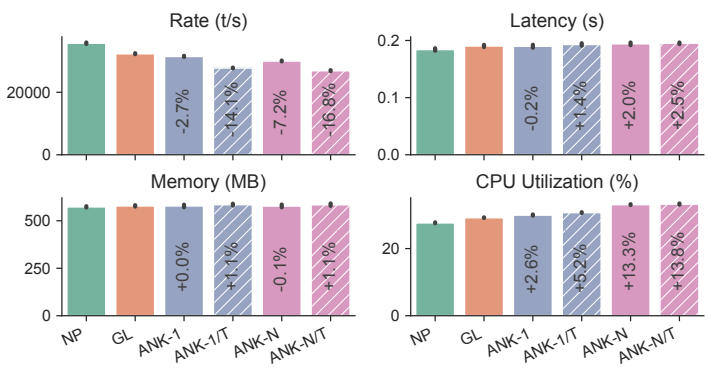


Figure C11: Performance - Smart Grid queries.

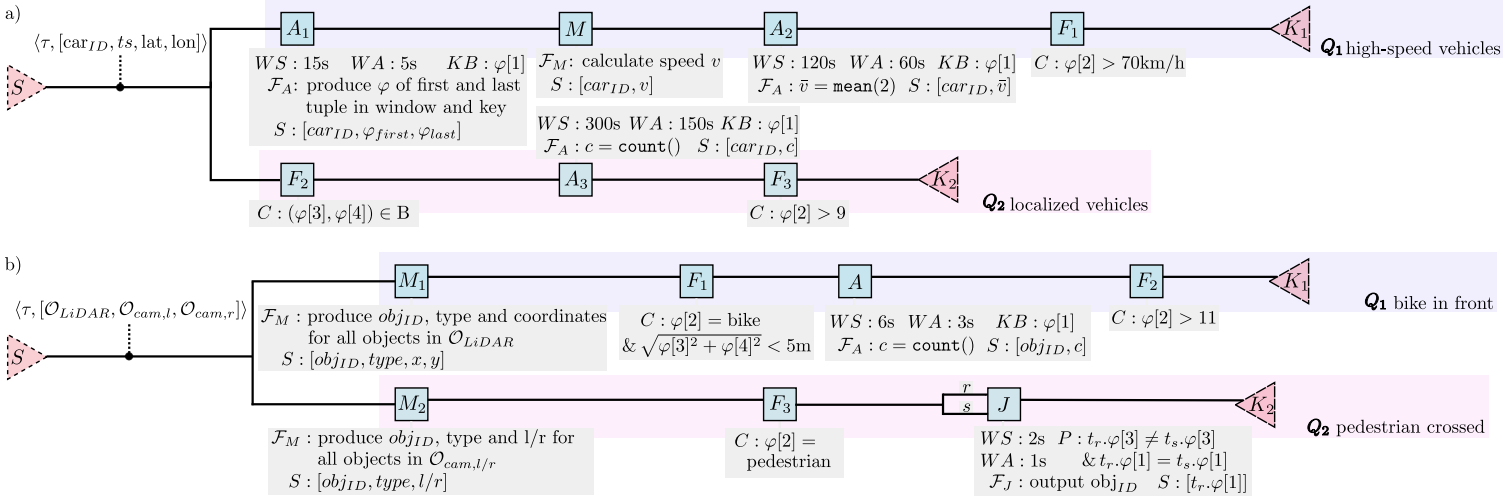


Figure C12: Real-world example queries: a) Vehicle Tracking queries. $\tau, \text{lat}, \text{lon}$ give the timestamp, GPS latitude and GPS longitude of the car with ID car_{ID}. b) Object Annotation queries. v_L, v_R are the camera images from the left- and right-facing camera modules; l is the LiDAR point cloud. *obj* is a concretely-bounded object within an image or a point cloud.

4 years around Beijing [214]. We employ 10046 traces of cars driving a full day each to simulate a large fleet driving simultaneously. Figure C12a shows the queries, fed tuples carrying the car ID, timestamp, and latitude/longitude. Q_1 calculates the immediate and average speed of the last two minutes per car, and forwards to K_1 tuples with average speed $\bar{v} > 70$km/h. Q_2 forwards events with more than 9 coordinates within area B, a region around Yuyuantan Park in Beijing, to K_2. This experiment is performed on an Odroid receiving the input data via 100MBit/s Ethernet. Sink tuples of Q_1 depend on around 30-160 tuples and sink tuples of Q_2 depend on 25-250 tuples. As shown in Figure C13, the performance of *Ananke* follows the same trend as before. The impact on rate and latency is small for ANK-1 (/T), at 2-4.5% more than GL and higher for ANK-N (/T), up to 18.3%. The larger provenance graphs of Q_1 and Q_2 cause *Ananke* to have higher resource requirements, with the memory utilization almost doubling in some cases and the CPU utilization jumping by up to 57% in the worst case (ANK-N).

Object Annotation queries These queries enrich an in-vehicle computer vision system based on LiDAR and two cameras. We use the Argoverse Tracking dataset [35], with 113 segments of 15-30s continuous sensor recordings of urban driving, plus 3D annotations of surrounding objects. The two queries, shown in Figure C12b, receive a stream of tuples carrying sets of annotations $\mathcal{O}_{LiDAR}, \mathcal{O}_{cam,L}, \mathcal{O}_{cam,R}$, of objects found by the vehicle's LiDAR and a left- and right-facing camera. These sets contain objects labeled with the type (e.g., "pedestrian"), 2D position, and a unique object ID. M_1 reproduces all objects found by the LiDAR, while F_1 forwards only bicycles found in front of the vehicle. A and F_2 then forward a tuple to K_1 if a specific bike was in front of the vehicle for more than 11 frames during a 6s window. In Q_2, only tuples referring to pedestrians are forwarded as two streams to a Join. If, during 2s, a certain pedestrian is found by both cameras, the pedestrian has crossed, and a tuple is forwarded to K_2. To simulate powerful, specialized vehicular hardware, this experiment was performed on the Xeon-Phi server. On average,

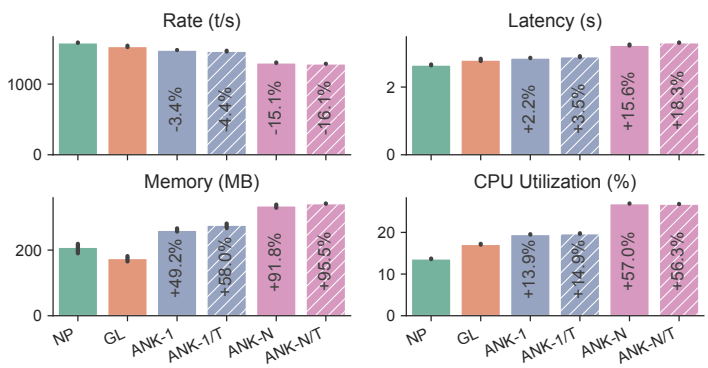


Figure C13: Performance - Vehicular Tracking queries.

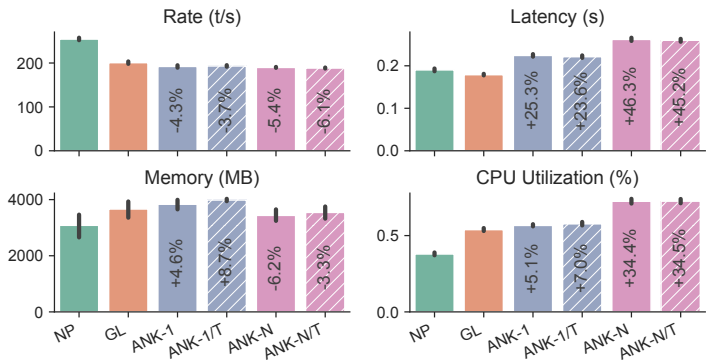


Figure C14: Performance - Object Annotation queries.

Q_1's sink tuples depend on 15-50 source tuples whereas Q_2's ones depend on 2. The tuples of these queries are much bigger than all previous use-cases, in the order of kilobytes instead of bytes. As evident by the performance of NP in Figure C14, these queries are much more demanding. For example, GL drops 21% in rate, mostly due to the large volume of provenance data transferred between the SPE tasks. *Ananke* has a small effect on the rate, causing a further drop of 3.9-6.6%. Latency is affected more, increasing by about 25% for ANK-1(/T) and 48% for ANK-N(/T), while remaining at small absolute values. Memory consumption does not change significantly compared to GL. The CPU grows for ANK-N(/T), similar to previous use-cases.

C6.2 Provenance Latency

We now study the provenance latency (Section C3) for ANK-1 and ANK-N for the Linear Road (LR), Smart Grid (SG), Vehicular Tracking (VT), and Object Annotation (OA) queries. The results are shown in Figure C15 in multiples of U (see Section C5), for vertices (SINK/SOURCE), edges (EDGE), and labels (SINK-L/SOURCE-L) of vertices. As shown, ANK-1 generally has a lower provenance latency than ANK-N. The main difference is that, while ANK-1 can output SOURCE almost immediately (and then store its ID to

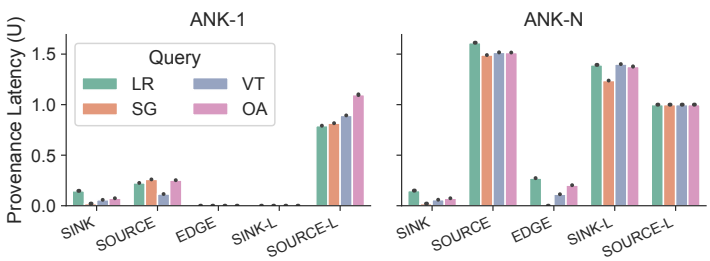


Figure C15: Provenance Latency in multiples of U.

not output it again), ANK-N delays the production of SOURCE by at least U to avoid duplicates. This delay propagates to SINK-L (see Section C5.2). Also, ANK-1 can immediately emit EDGE and SINK-L without any delay (see Algorithm C1). Both variants have low SINK latency as those are safe to emit immediately upon arrival of a sink tuple. The variance is close to zero, as the frequency of watermark updates, the only execution-dependent feature of provenance latency, does not change between repetitions.

C6.3 ANK-1 vs. ANK-N Trade-offs

Here, we compare ANK-1 and ANK-N for different data characteristics and query configurations. The experiments, run on the Xeon-Phi server, use synthetic queries in which Sources feed *Ananke* (non-transparent) with pre-populated provenance graphs.

Figure C16 shows the performance for different provenance sizes (x-axis) and overlaps (bar colors). The former is the number of source tuples each sink tuple depends on, the latter is the percentage of shared provenance between subsequent sink tuples. ANK-1 has better performance and lower resource requirements than ANK-N, due to the simpler algorithm and single-task deployment. Both ANK-1's and ANK-N's performance drops as the provenance size increases, as more data is maintained and transferred between tasks. Larger provenance overlaps result in slightly better performance since fewer source vertices and labels are emitted.

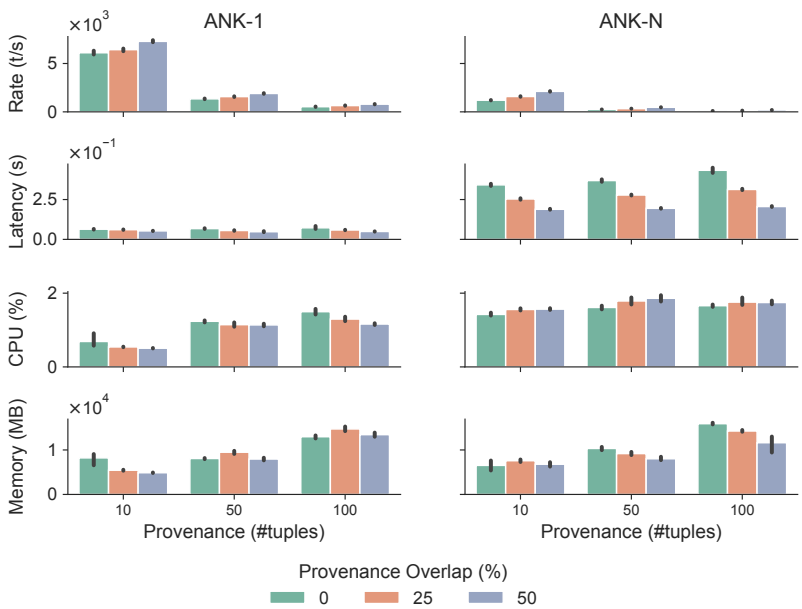


Figure C16: Performance of the synthetic query for different overlaps and provenance sizes.

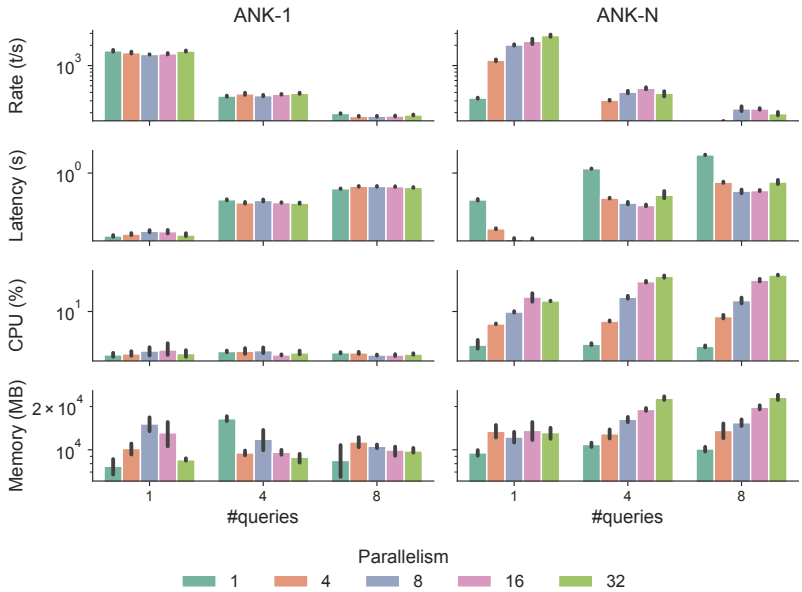


Figure C17: Performance of the synthetic query for different parallelisms and #queries (logarithmic scale).

Figure C17 studies ANK-N's ability to take advantage of the scalability features of the SPE. The x-axis is the number of queries feeding data to *Ananke*, and the bars refer to different parallelism values of *Ananke*. The provenance size is 50, and the overlap 25%. As shown (in log scale), ANK-N outperforms ANK-1 for parallelism 4 or higher since ANK-1 does not support parallel execution. ANK-N's resource consumption increases with parallelism, making it better suited for use-cases with more available resources. For both ANK-1 and ANK-N, a higher number of queries results in a drop in performance, caused by the increased data flow.

C6.4 Comparison with On-demand Techniques

ANK-1 and ANK-N are not the only ways to achieve the goals of Section C3. Here, we compare *Ananke* with ad-hoc alternatives relying on existing systems (external to the SPE) to produce the provenance graph on-demand. These alternatives can seem appealing due to their additional features, e.g., persistent storage of backward provenance. Thus, a comparison with them is crucial to understand the properties of the fully-streaming approach provided by *Ananke*.

We study alternatives based on database systems with varying performance and safety guarantees. The first, SQL-P, relies on an established relational database (PostgreSQL [159]), the second, SQL-I, on a fast, self-contained relational database running *in-memory* (SQLite [179]), and the third, NoSQL,

on a non-relational database (MongoDB [141])[C7]. The SQL implementations adhere strictly to the goals of Section C3, whereas NoSQL follows a best-effort principle, without strict ordering guarantees (due to the concurrent accesses by several threads). In contrast with *Ananke*, the alternatives produce the (streaming) provenance graph on-demand. A thread polls the database periodically and performs the data transforms. We evaluate an *aggressive* (suffix /A) polling strategy (as frequently as possible), and a *relaxed* (suffix /R) one (polling every second).

We compare ANK-1 with the above alternatives for two real-world experiments (Smart Grid and Vehicle Annotation queries), as well as for a synthetic one. In addition to previous performance metrics, we also study the *delivery latency* of the provenance graph, to assess the benefits and drawbacks of dif-

[C7]A graph database (Neo4J [143]) seems like an obvious choice for provenance data [190], but our preliminary experiments indicated it performed much worse than the alternatives in our streaming applications and is thus not presented here.

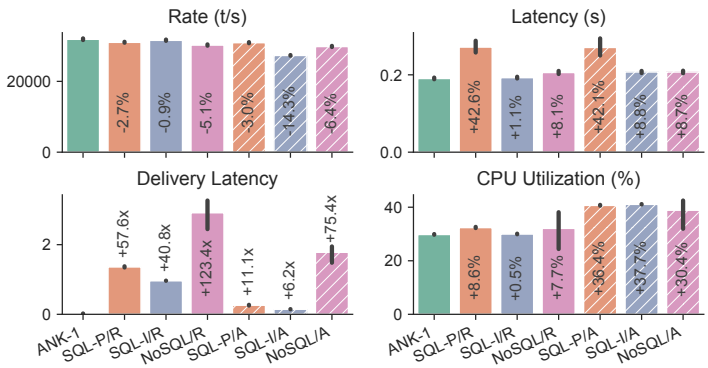


Figure C18: Smart Grid: Performance comparison between *Ananke* and on-demand implementations.

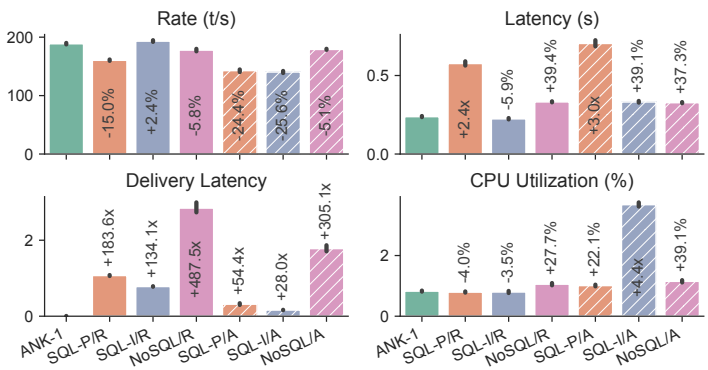


Figure C19: Object Annotation: Performance comparison between *Ananke* and on-demand implementations.

ferent polling strategies. This metric expresses the delay (in wall-clock time) between a graph component (vertex, edge, "expired" label) being ready to be delivered and actually being delivered. We do not study memory consumption, as the usage of external systems with different storage mechanisms creates a non-uniform measurement environment.

Figure C18 and Figure C19 present the performance of the Smart Grid and Object Annotation queries, respectively. The text inside the bars indicates the relative difference from ANK-1. The performance of the on-demand implementations ranges widely, with relaxed polling (/R) having better rate, latency, and CPU but more than one order of magnitude higher delivery latency. Aggressive polling (/A) lowers the delivery latency but severely degrades the other metrics, in most cases. SQL-I/A achieves the lowest delivery latency (up to 6.2x higher than ANK-1), whereas SQL-I/R has the least impact on the original queries (slightly better rate and latency than ANK-1, at the price of 28x the delivery latency and 4.4x the CPU). SQL-I's implementation closely resembles ANK-1, temporarily maintaining in-memory only unsent graph components, instead of persisting all backward provenance data like SQL-P and NoSQL. This similarity in SQL-I's and ANK-1's implementations explains their similarity in performance. NoSQL's best-effort strategy results in a relatively low impact on the original queries but multiple orders of magnitude higher delivery latency than ANK-1, in most cases. SQL-P performs between SQL-I and NoSQL, with lower delivery latency than NoSQL but a much higher impact on the other metrics. This is expected, as SQL-P persists the backward provenance (unlike SQL-I), while also strictly adhering to the goals of Section C3, in contrast with NoSQL. Both experiments indicate that ANK-1 significantly outperforms all studied alternatives in most or all metrics.

Table C2 presents the aggregated results of the synthetic experiment, having a setup[(C8)] similar to Figure C16, with a provenance overlap of 0% and provenance sizes 10-100. The relative performance is comparable to the real-world experiments, with ANK-1 outperforming the alternatives overall. SQL-I seems to outperform ANK-1 in rate and latency but a detailed examination of the experimental data indicates that SQL-I can only sustain 40% of the

[(C8)] For a fair comparison with the on-demand implementations, in this experiment ANK-1 is writing the graph to disk, and thus has a lower performance than in Figure C16.

Table C2: Performance comparison between *Ananke* and on-demand implementations in the synthetic query.

	ANK-1	SQL-P		SQL-I *		NoSQL *	
		/R	/A	/R	/A	/R	/A
Rate (t/s)	131.3	54.43	54.80	211.4	209.72	19.81	19.75
Latency (s)	1.03	2.86	2.82	0.62	0.70	16.19	16.61
Deliv. Latency (s)	0.07	2.97	0.78	25.10	15.24	27.63	27.29
CPU (%)	1.18	0.97	1.30	1.75	1.97	0.92	0.89

* The values for SQL-I and NoSQL do not reflect the steady-state performance, which is significantly lower. Refer to the text for more details.

measured rate without exhausting the memory of the system. As, in contrast to ANK-1, the on-demand alternatives lack a back-pressure mechanism, they can fetch backward provenance from the SPE faster than they can process it. This leads to a continuously-increasing provenance backlog (illustrated by the high delivery latencies). For an in-memory database like SQL-I, this is unsustainable[(C9)]. While backpressure could be added manually to the studied alternatives, this would be "reinventing the wheel" as such mechanisms are available out-of-the-box in *Ananke*, running inside the SPE.

Among the studied on-demand alternatives to *Ananke*, SQL-I performs best but is still disadvantaged by not being streaming-oriented. In contrast, ANK-1 has a much better sustained performance and does not have to maintain "raw" provenance. This enables the immediate forwarding of the forward provenance graph stream to a secondary ingesting system without keeping unnecessary state, saving space and computational resources.

Evaluation summary

Ananke has similar overheads to the state-of-the-art in backward streaming provenance while offering live, forward rather than simply backward provenance. Compared to [151], the best-performing implementations of *Ananke* incur less than 5% drop in the rate in all use-cases and less than 3% increase in latency in all but one use-case. The evaluation shows that *Ananke* is suitable in deployments of real-world applications requiring both efficient processing and live provenance capture. Alternative existing systems fall short in providing the graph timely and in a sustainable fashion suited to the data streaming paradigm.

C7 Related Work

Data provenance, extensively studied in databases [38], [48], [94], only recently started being in focus in data streaming. Early such work [191] focuses on coarse-grained data stream dependencies. A finer-grained approach, in [195], produces time intervals which *may* contain provenance tuples. [98] focuses on minimizing the storage requirements of streaming provenance but produces approximate results and lacks support for some native operators (e.g., Join).

To the best of our knowledge, *Ananke* is the first to deliver live forward provenance. [50] presents a system for debugging streaming queries with integrated provenance capture and visualization. However, it focuses on one SPE [72] and slices of the execution (with the option to lazily create the complete query provenance). It requires users to declare relationships between input and output tuples and does not distinguish expired tuples. Likewise, StreamTrace [20], targeting the Trill SPE [34], assists the development and debugging of queries with data visualization, relying on provenance through instrumented operators and ad-hoc query rewriting. *Ananke*'s provenance graph

[(C9)] NoSQL suffers from the same issue; however, in this case, it is less critical since NoSQL does not need to maintain its state in-memory.

allows creating similar visualizations for the end-to-end system provenance. GeneaLog is the state-of-the-art technique and framework for fine-grained provenance in streaming applications. It records and traces the provenance metadata while incurring a small, constant per-tuple overhead. In this work, we extend GeneaLog to support out-of-order data and use it as the provider of backward provenance for *Ananke*. Ariadne [76] is a similar framework for fine-grained streaming data provenance using instrumentation. However, it requires variable-length annotations and needs to temporarily store all alive source tuples (including those that do not contribute to any sink tuples) until they become expired. The authors hypothesize the use of static query analysis to discern alive and expired tuples, but without further details. As discussed in Section C3, *Ananke* can be adjusted for use with Ariadne or any streaming backward provenance framework.

C8 Conclusions

We presented *Ananke*, a framework to extend streaming tools for backward-provenance and to deliver a live bipartite graph of fine-grained forward provenance. *Ananke* provides users with richer provenance information, not only specifying which source tuples contribute to which query results, but also whether each source tuple can potentially contribute to future results or not. This distinction can help analysts prioritize the inspection of the large volumes of events commonly observed when monitoring CPSs. We formally prove *Ananke*'s correctness and implement two variations (available in [75]) in Flink. Through our thorough evaluation, we show that *Ananke* incurs small overheads while being able to outperform alternatives relying on tools external to the SPE. Future work can address (1) finer-grained debugging and exploration by expanding *Ananke*'s live provenance to include intermediate tuples produced by query operators, also indicating whether such tuples could contribute to future results, and (2) exploring how *Ananke*'s theoretical foundations about alive/expired tuples can be used in fault-tolerant stream processing.

Chapter D

Data Selection via Dynamic Forward Provenance

Bastian Havers, Marina Papatriantafilou, Vincenzo Gulisano

The following is an adapted version of the work under submission as *"Nona: A Framework for Elastic Stream Provenance"*. Any changes serve only to retain the consistency of this thesis.

Abstract

Forward Provenance for streaming queries run by distributed and parallel Stream Processing Engines gives fine-grained insights on input-output data dependencies enabling, e.g., precise debugging and smart data selection. State-of-the-art provenance frameworks, though, build on an assumption that is unrealistic for distributed systems like Vehicular Networks and Smart Grids, namely, that the whole set of queries in need of provenance is known in advance and static. In real-world use cases, queries are continuously added, removed, and modified over time by both data analysts and SPE systems themselves.

Motivated by the lack of solutions for the forward provenance of dynamic sets of queries, we introduce a novel framework, named Nona, for parallel and distributed streaming queries. We formalize the notion of forward provenance for evolving query sets and prove it is possible to extend the same guarantees the state-of-the-art offers for static query sets. Our evaluation shows that Nona can cope with adaptations to changes in query sets with sub-second responsiveness; moreover, it incurs negligible overheads compared to the state-of-the-art, during the periods in which a query set does not undergo changes.

D1 Introduction

Systems such as Vehicular Networks and Smart Grids [91] leverage the stream processing paradigm [3], [180] and Stream Processing Engines (SPEs) [8], [10], [29] for continuous, low-latency analysis of high-volume and fast-paced streaming data.

In data streaming applications (or streaming *queries*), *forward* provenance [153] seamlessly connects analysis results to the input data causing them. The resulting input-output dependencies then help debug applications, understand how results came to be, and enable efficient selection of relevant input data for, e.g., further processing, transmission, or storage.

State-of-the-art forward provenance frameworks [153] ensure *three key properties* while delivering input-output dependency graphs. In particular, they deliver each input, output, and connecting edge **(i)** logically ordered by cause-and-effect and **(ii)** exactly once, without duplicates, no matter the number of queries processing data in a distributed and/or parallel fashion. By deduplicating data, they can reduce the volume of data forwarded or stored in distributed systems for which communication and storage are limited or costly (e.g., Vehicular Networks and Smart Grids). Also, they **(iii)** continuously distinguish, over time, which inputs could still contribute to new outputs (i.e., which vertexes could have more edges connected to them) from those that are *final*. This distinction helps differentiate which parts of the graph are ready for further processing, since processing parts that are still evolving may produce incomplete results or require later corrections.

Motivating Challenge

State-of-the-art forward provenance frameworks build on an assumption that, practically, makes them unusable in distributed systems like Vehicular Networks

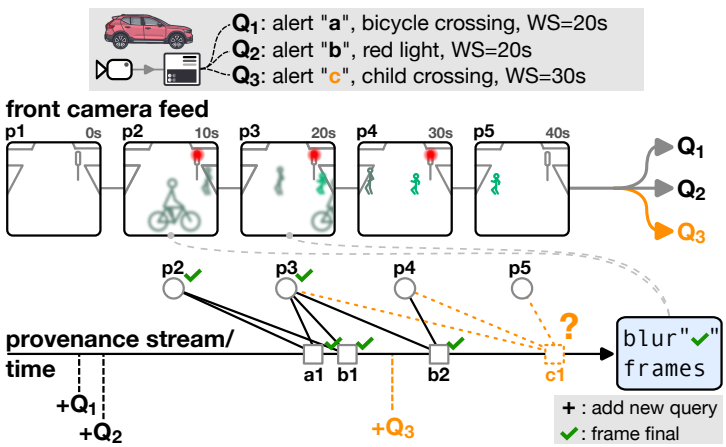


Figure D1: Running example: Dynamic in-vehicle query deployment with provenance (WS = window size).

and Smart Grids. Namely, that the whole set of queries for which provenance is to be provided is known in advance and static. This is not realistic: queries are added, removed, and modified over time by both data analysts and SPE systems themselves (e.g., when moving a query to rebalance the workload distribution of a set of nodes). As the next example shows, dynamic query behavior brings key novel challenges.

Running Example (Figure D1) - Scenario

Queries Q_1, Q_2 start processing a vehicle's camera frames $p1$, $p2$,... (framerate 0.1Hz) from a buffer. Q_1 produces alert a on crossing bicycles within a window of 20s; Q_2 produces alert b on red lights visible for 20s. The forward provenance for Q_1 and Q_2 is generated as a stream post-processed by an external component that blurs peoples' faces in frames of the input buffer that contributed to an alert and are marked as *final*, to preserve people's privacy when such data is later processed by other systems. If Q_1 and Q_2 are the only queries ever running, the blurring does not risk falsifying results from other queries. Suppose, though, that after images $p2, p3$ were blurred a new query Q_3, producing alerts c on children crossing within 30s, is added. When Q_3 now ingests frames from the input buffer, will it be able to detect a child from a blurred face in $p3$-$p5$? Additionally, can Q_3 be affected by concurrency issues if it processes a frame while such a frame gets blurred?

In real-world applications, ad-hoc solutions to dynamic modifications of the running queries with no formal specification of which properties a forward provenance framework enforces are not usable in practice. One could, e.g., hinder Q_3 from processing tuples that are final in the graph - but where will a cutoff be set in such a dynamic system if Q_3 is to access as much data as possible? And, should Q_2 be removed, how can an analyst know whether Q_2's latest inputs that led to an alert will be delivered in the dependency graph? In the state-of-the-art, no known streaming provenance framework exists that provides properties (i)-(iii) while accommodating a *dynamic* (evolving) set of queries.

Contribution

We pose the following question:

"How can forward provenance be provided for the dynamic *case, i.e., for sets of queries not known a priori and evolving over time?"*

Referring to the type of forward provenance in question as *Dynamic Query Set Forward Provenance* (D$\mathbb{Q}$-FP), and to that of a static set as *Static Query Set Forward Provenance* (S$\mathbb{Q}$-FP), we make the following contributions:

- we formalize the problem of D$\mathbb{Q}$-FP;
- we analyze and evaluate the trade-offs in different ways fulfilling D$\mathbb{Q}$-FP's requirements;

- we design and implement the first framework for D$\mathbb{Q}$-FP, Nona, that allows to add or remove queries at runtime while supporting distributed and parallel execution;
- lastly, we conduct a thorough evaluation of Nona using Apache Flink [29] and Kafka [9], with real-world data and benchmarks, showing performance equal to S$\mathbb{Q}$-FP state-of-the-art on static and fast adaptation to changing query sets.

All required code is open-sourced at [89] for reproducibility. To the best of our knowledge, our work is the first to formalize and present an algorithmic implementation for D$\mathbb{Q}$-FP.

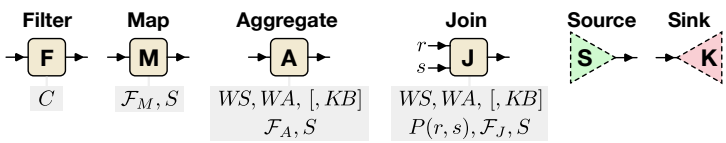
Organization

Section D2 presents preliminary data streaming and provenance concepts; Section D3 details the problem definition and the goals of the work; Section D4 discusses in-depth theoretical challenges; Section D5 presents an algorithmic solution to the work's goals, which is evaluated in Section D6. Section D7 discusses related work, and Section D8 summarizes and concludes.

D2 Preliminaries

D2.1 Elementary Stream Processing Notions

We follow the DataFlow model [3]: A data *stream* is a series of *tuples* $t = \langle \tau, \phi \rangle$ carrying metadata (a timestamp $t.\tau$ and possibly further attributes) and a multi-attribute payload $t.\phi$. A streaming *query* consists of *sources*, *operators*, and *sinks* composing a directed acyclic graph (DAG). Sources forward streams of *source tuples*, e.g., from an *external source* (e.g., a physical sensor) to operators that process tuples. Any stream can be multiplexed to an arbitrary number of downstream operators or sinks. Once a sink has finished processing a *sink tuple* t (e.g., forwarding t to an external application or writing it to a file), we say the sink has *observed* t. Sources and operators can create tuples. τ is set by the source to the time a corresponding event occurred, the *event time*. Whenever an operator O produces a tuple t, $t.\tau$ is set according to O's semantics, and $t.\phi$ according to user-defined functions.

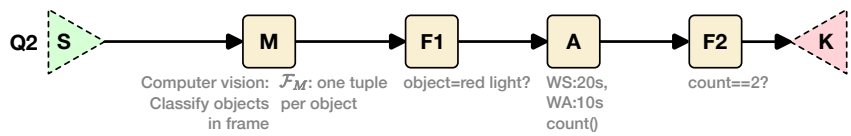


Common operators supported by SPEs [7], [8], [10], [29] are *Filter*, *Map*, *Aggregate*, and *Join*. Map and Filter are *stateless*, and process tuples 1-by-1. **Filters** discard input tuples or forward them as-is depending on whether a condition C holds. **Maps** use a user-defined function $\mathcal{F}_M$ to create an arbitrary number of tuples t_O per input tuple t_I, with $t_O.\tau = t_I.\tau$.

Stateful operators ingest delimited groups of tuples called *windows*. We focus on *time windows* that retain tuples as defined by an operator's *window size WS* and *window advance WA*. A window with, e.g., *WS*, *WA* set to 10 and 5 minutes will thus cover intervals [0:00,0:10), [0:05,0:15), etc., where every interval denotes a *window instance* (we distinguish between window and its instances only where ambiguities could arise). If $WS > WA$, consecutive window instances can overlap and tuples can *fall* into several window instances.

The **Aggregate** stateful operator is defined by: (1) *WS*, *WA*: the window size and advance, (2) *KB*: an optional key-by function to maintain separate (yet aligned) window instances for different key-by values, and (3) $\mathcal{F}_A$: a function to aggregate the tuples in one window instance into the ϕ attribute of an output tuple. When an output tuple is created for a window (and a key-by value, if *KB* is defined), its timestamp is set to such window's right boundary [7], [10], [29]. A **Join** matches tuples from two input streams, r and s. It keeps separate windows for r and s tuples, which share *WS* and *WA*. Each pair of r and s tuples (optionally sharing a common key *KB*) are matched for every pair of windows covering the same event-time period they fall in. The Join operator is defined by: (1) *WS*, *WA*, (2) *KB*, (3) $P(t_r, t_s)$: a predicate for pairs of tuples from r and s, and (4) $\mathcal{F}_J$: a function to create the ϕ attribute of the output tuple, for each pair of tuples for which $P(t_r, t_s)$ holds (and which optionally share the same *KB* key). The timestamp τ of such output tuple is set to the shared right boundary of the aligned windows of r and s.

RUNNING EXAMPLE (FIGURE D1) - QUERY DAG



The above figure shows query Q_2 from the running example (Figure D1) that detects red lights in time windows of twenty seconds.

D2.2 Watermarks and Time Progression

Parallel operator instances or multiple asynchronous operators can result in tuples being fed to a downstream operator out of timestamp order. Hence, for stateful operators, receiving a tuple with a timestamp greater than some window's right boundary does not imply later tuples will no longer fall into said window. To prevent this disorder from resulting in incorrect results, a partial order on streams [99] is provided by *watermarks* in both early and state-of-the-art SPEs [29], [99]:

Definition D1. *The value of the watermark W_O^ω of some operator O denotes that after the point ω in wall-clock time (or simply: time), O will only receive tuples t with $t.\tau \geq W_O^\omega$.*

Note that we include the superscript ω only where it cannot be induced from

context. Sources periodically emit watermarks as special tuples downstream[D1]. Receiving a watermark, an operator O updates and forwards a local watermark-variable W_O to $\min_{I \in \mathbb{I}} \big(\max(W_I)\big)$ downstream, where $\mathbb{I}$ is the set of O's input streams and $\max(W_I)$ is the largest watermark received from input stream I. For an Aggregate or Join Ω, whenever the watermark advances from W_Ω^ω to $W_\Omega^{\omega'} > W_\Omega^\omega$, an output tuple is created for each window instance maintained by Ω that has a right boundary less than or equal to $W_\Omega^{\omega'}$, said window instance is discarded, and $W_\Omega^{\omega'}$ is forwarded.

D2.3 Static Query Set Forward Provenance: SQ-FP

Following the definitions from [153], SQ-FP allows for each source tuple t_S of a static set of queries to retrieve the sink tuples t_K to which t_S *contributes* (and, vice versa, to retrieve t_S from t_K), where the *contributes* relation is defined as:

Definition D2. *Let t, t^* be tuples in a query execution; we say that t* contributes directly *to t^* if some operator emits t^* as a consequence of processing t, also written as $t \rightarrow t^*$. If $t \rightarrow t' \rightarrow t'' \rightarrow \ldots \rightarrow t^*$, then we say that t* contributes *to t^*: $t \rightsquigarrow t^*$ (thus, if $t \rightarrow t^*$, then $t \rightsquigarrow t^*$).*

SQ-FP delivers these relations as a stream of graph elements representing the *forward provenance graph* or simply *provenance graph*. Vertexes (i.e., sink and source tuples) are (i) ordered by cause and effect, and delivered before any edge (contributes-to relation) connecting them, (ii) *deduplicated*, i.e., each source tuple appears exactly once, no matter the number of sink tuples it contributes to, and (iii) labeled to indicate, for each point in time ω, which tuples are *expired*:

Definition D3. *Let t, t^* be tuples in a query execution. At time ω, t is* active *if it can still contribute directly to some tuple produced by some operator. At time ω, t is* alive *if it is* active *or if there is at least one active tuple t^* such that $t \rightsquigarrow t^*$. Otherwise, t is* expired.

In a provenance graph, since no further edges will connect to a vertex v expired at ω (i.e., v's neighborhood is static after ω), we say v is *final*. A tuple expiration threshold $T_\mathbb{Q}^\omega$ for any ω (s.t. after time ω, any tuple with $t.\tau < T_\mathbb{Q}^\omega$ is expired) can be calculated for a static set of queries $\mathbb{Q}$ as follows [153], where for simplicity we assume one sink per query:

$$T_\mathbb{Q}^\omega = \min_{Q \in \mathbb{Q}} \left(W_Q^\omega - U_Q\right), \tag{4.1}$$

where W_Q^ω is the watermark of the sink of query Q at time ω, and the *maximum aliveness constant* U_Q is defined as follows:

$$U_Q = \max_{p \in \mathbb{P}_Q} \sum_{j \in p} WS_j, \tag{4.2}$$

[D1]These are also called *in-band* watermarks; for a discussion of alternative watermarking schemes see for example [99], [125].

where $\mathbb{P}_Q$ is the set of all paths p from any source in Q to Q's sink, and j indexes all operators along path p. U_Q is the maximum timespan a tuple may remain alive in query Q [153].

D2.4 Backward Provenance

As shown in [153], S$\mathbb{Q}$-FP or forward provenance can be obtained from a stream of *backward provenance* [76]: Backward provenance connects each sink tuple t_K to its contributing source tuples t_S (see Definition D2). Differently from forward provenance, though, a backward provenance stream is not deduplicated and does not distinguish expired tuples. The backward provenance stream allows to access source tuples contributing to a sink tuple by metadata such as pointers [150] or IDs identifying source tuples in a dedicated buffer [77].

D3 Problem Formalization

D3.1 System Model

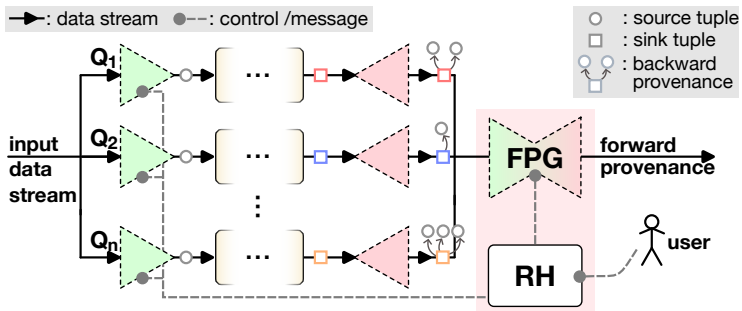


Figure D2: System model: n queries feeding backward provenance to the Forward Provenance Generator operator (FPG). Users interact with the system via the Request Handler (RH).

We consider systems in which queries are running in parallel and independently. For ease of exposition, we assume the queries are being fed one external input stream. Shown in Figure D2, every query Q in the query set has a sink generating backward provenance for every sink tuple t_K, retrievable by the procedure GETBPROVENANCE(t_K). All sinks forward the backward provenance as well as watermarks to a streaming operator FPG (Forward Provenance Generator) generating the forward provenance. The additional Request Handler (RH) is interfaced by a user to request to add an additional query to the n already running queries, or to remove one of them.

D3.2 Definitions

To formulate our aims in detail, we introduce in this section some fundamental definitions related to D$\mathbb{Q}$-FP.

Definition D1 (Dynamic Query Set)**.** *The* dynamic *query set at wall time* ω*,* $\mathfrak{Q}(\omega)$*, is an evolving set of queries that are fed tuples from one or more external sources, and that can produce results.* $\mathfrak{Q}(\omega)$ *undergoes* transitions *in which a single query is* added *or* removed*.*

In the following, ω will be omitted if its value is not clear from context. Specifically, a query is fed source tuples and can produce results and watermarks if and only if it is part of $\mathfrak{Q}$. Thus, once a query is removed from $\mathfrak{Q}$, no more tuples are fed to it and it can no longer produce results or watermarks.

For clarity of presentation details, we consider the initial query set to be empty; the problem formulation and methodology are applicable also when this condition does not hold.

Transitions occur upon *requests* R_i issued by users:

Definition D2 (Requests and Transitions)**.** *Requests* R_i *trigger one of the following:*

- $R_i = +Q_i$*:* addition transition
- $R_i = -Q_j\,(j<i)$*:* removal transition

We denote by s_i *the time point at which* R_i *is issued, and by* f_i *the time point when the transition is finished and* $Q_{i/j}$ *starts/stops being a member of* $\mathfrak{Q}$*.*

Note that we regard every added query as unique, even though their DAG may be identical. For removals, condition $(j < i)$ applies, as only queries added earlier can be removed.

As an example, if the dynamic set is comprised of the queries $\mathbb{X}$ and then a request $R_k = +Q_k$ is issued, we write

$$\mathfrak{Q}(s_k) = \mathbb{X} \xrightarrow{R_k=+Q_k} \mathfrak{Q}(f_k) = \mathbb{X} \cup \{Q_k\}$$

and analogously for a removal request. For ω in the transition period $[s, f]$, the queries in $\mathfrak{Q}(\omega)$ are continuously fed source tuples and produce sink tuples. Note that Definition D2 effectively allows the arbitrary modification of a running query Q_l by either adding a modified version $Q_m, m > l$ to $\mathfrak{Q}$ first and then removing Q_l, or by the reverse procedure[(D2)].

Now, we formally define the *Forward Provenance Graph*:

Definition D3. *Denoting by* $v(t)$ *a vertex corresponding to a tuple* t*, the* Forward Provenance Graph $\mathcal{P}(\omega)$ *of a dynamic query set* $\mathfrak{Q}(\omega)$ *at time* ω *in an execution is a graph containing*

- *a sink vertex* $v(t_K)$ *per sink tuple* t_K *that has been observed by any sink* K_Q *at or before* ω*, where* Q *was or is in* $\mathfrak{Q}$*;*
- *a source vertex* $v(t_S)$ *per source tuple* t_S *for any sink vertex* $v(t_K)$ *where* $t_S \rightsquigarrow t_K$ *(see Definition D2);*

[(D2)] In the first procedure a tuple may be processed by both queries if they are part of $\mathfrak{Q}$ for some time simultaneously, while in the second a tuple may be processed by neither if it falls into a gap between removal and addition.

- *an edge* $e=(v(t_S), v(t_K))$ *per pair* (t_S, t_K) *where* $t_S \rightsquigarrow t_K$*;*

- *a label per node* $v(t)$ *where* t *is expired (see Definition D3).*

Note that it is insufficient if a sink tuple was only delivered to a sink, but the sink must have *observed* said tuple, where the definition of *observe* (see Section D2.1) in the first bullet-point depends on the specific implementation of the streaming sink[(D3)].

D3.3 Main Goal and Requirements

Goal of Nona

To produce, for the execution of the *dynamic* query set $\mathfrak{Q}$, the forward provenance stream $S_{\mathcal{P}}$ carrying

- one vertex tuple $t^V_{S/K} = \langle \tau_V, [ID_{S/K}, t_{S/K}] \rangle$ per source or per sink vertex in $\mathcal{P}$,
- one edge tuple $t^E = \langle \tau_E, [ID_S, ID_K] \rangle$ per edge $(v(t_S), v(t_K))$ in $\mathcal{P}$ s.t. $\tau_E > \tau_V$ for $V = S, K$,
- one label tuple $t^L = \langle \tau_L, [ID_{S/K}] \rangle$ per expired source/ sink vertex $t_{S/K}$ in $\mathcal{P}$ s.t. $\tau_L > \tau_E$ for any edge adjacent to $t_{S/K}$.

Requirements for Nona

R1 Completeness: Containing in $S_{\mathcal{P}}$ all vertices and edges from $\mathcal{P}$

R2 Expiration Promise: marking expired tuples and never reprocessing them

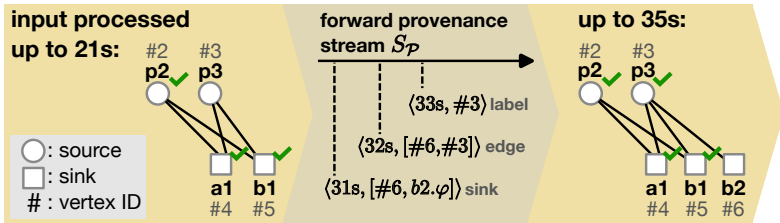
R3 Ordering and Uniqueness: ensuring the ordering and uniqueness constraints formulated in the main goal

R4 Responsiveness: enabling high responsiveness to user requests

Nona's goal extends that of SQ-FP defined in [153], by incorporating the dynamic nature of $\mathfrak{Q}$ that leads to an evolving forward provenance graph $\mathcal{P}$. The following example shows how the forward provenance stream $S_{\mathcal{P}}$ presents the forward provenance graph $\mathcal{P}$:

[(D3)] E.g., a sink may have observed a tuple once it has been written to disk, or once an external database has acknowledged receiving the sink tuple.

Running Example (Figure D1) - $\mathcal{P}$ and $S_{\mathcal{P}}$



The provenance graph of Figure D1 at two time points: sink vertices a, b are linked to the contributing source vertices p. Each vertex appears at most once, and vertices of expired tuples are marked as final (see checkmarks). Between 21s and 35s, new components are delivered in the forward provenance stream $S_{\mathcal{P}}$, adding sink vertex $b2$, its edge to vertex $p3$, and marking $p3$ as final. $b2$ is not final yet, as source vertex $p4$ (contributing to $b2$) and the connecting edge were not yet delivered on $S_{\mathcal{P}}$.

The goal is broken down into four requirements: (R1) The dynamicity of the query set complicates to generate the *complete* edge and vertex set contained in $\mathcal{P}$, in the face of query removals that may lead to missing elements in $S_{\mathcal{P}}$ (e.g., for some sink vertex not all contributing source vertex tuples are produced if the query is removed prematurely) and query additions (that abruptly generate a new source of provenance graph elements). (R2) While the asynchronous nature of queries, operators and data forwarding in stream processing boosts performance, it also requires careful synchronization to guarantee that tuples marked as expired are and remain expired in the face of a dynamic query set, while simultaneously desiring a non-blocking implementation. It must be ensured that no query may ever process an expired tuple, nor must tuples be labeled as expired with unnecessary latency (the expiration threshold in Equation 4.1 provides only an upper bound on tuple expiration, incurring a base latency depending on the window alignment in the queries. Late updates of the expiration threshold incur additional latency overheads). (R3) The ordering and uniqueness constraints from Definition D3 must be met to provide a forward provenance stream identical to that of SQ-FP (see [153]). (R4) The system must provide a high level of responsiveness to user requests for changes in the query set, without halting query execution for addition or removal.

On top of these requirements, to be a streaming-based framework that receives and processes watermarks from the individual queries for which forward provenance is provided. As earlier work for SQ-FP has shown, approaches using databases incur higher latencies and overheads [153].

In the following, we show how to fulfill these requirements and how our implementation of Nona achieves them.

D4 Guaranteeing Completeness and the Expiration Promise

Here, we investigate the challenges from providing D$\mathbb{Q}$-FP for the dynamic query set $\mathfrak{Q}$, and present solutions to meet the conceptual requirements R1-R2 from Section D3 (keeping discussions of the algorithmic implementation and system requirements R3 and R4 to Section D5). We assume that there is a single input stream forwarded to all queries in $\mathfrak{Q}$ (as sketched in Figure D2), both for ease of exposition and as sharing input tuples between queries exacerbates the need for careful analysis (cf. Figure D1).

While an added query becomes part of $\mathfrak{Q}$ at f_i, additional deployment time may be spent by the SPE before the query can begin processing tuples. Thus, for discussions we include an additional time point per transition, g_i with $g_i > f_i > s_i$:

request	$\mathbf{s_i}$	$\mathbf{f_i}$	$\mathbf{g_i}$
$R_i = +Q_i$	R_i issued	$\mathfrak{Q} \leftarrow \mathfrak{Q} \cup \{Q_i\}$	Q_i ingests tuples
$R_i = -Q_j$	R_i issued	$\mathfrak{Q} \leftarrow \mathfrak{Q} \setminus \{Q_j\}$	-

D4.1 R1: Completeness

To provide completeness, any edge and vertex of the provenance graph $\mathcal{P}$ must occur in $\mathcal{S}_\mathcal{P}$. Following Definition D3, once a sink tuple t_K has been *observed* by a sink, t_K itself, any source tuple t_S contributing to t_K, and edges connecting t_S, t_K are part of $\mathcal{P}$ and thus must occur as elements of $\mathcal{S}_\mathcal{P}$.

D4.1.1 Additions: $R_i = +Q_i$

Q_i will begin processing input tuples at g_i. Once a result from Q_i at some point at or after g_i has reached the sink K_{Q_i} and has been forwarded to the FPG (see Figure D2), it has been *observed* in accordance with Definition D3, Then, its forward provenance is part of $\mathcal{P}$ and must be contained in the graph stream $\mathcal{S}_\mathcal{P}$ to achieve completeness. As K_{Q_i} provides backward provenance (see Section D3.1), and thus access to all sink tuples contributing to a source tuple, the FPG has access to all sink and source tuples in $\mathcal{P}$ and can thus provide completeness.

D4.1.2 Removals: $R_i = -Q_j$

Removing query Q_j means there will be a final sink tuple from it that can be considered as observed, t_f. To ensure completeness, thus all sink tuples and their Backward Provenance from Q_j have to be processed by the FPG *up to and including* t_f. Once this is done, Q_j can safely be considered as removed. As there are no guarantees on how long it takes before a tuple emitted by a sink is processed by the FPG, the FPG can not determine by the passage of clock time which sink tuple is the last from Q_j. Thus, the final tuple is marked with the *marker tuple*:

Definition D1. *The* marker tuple *is a watermark with value infinity and is the last watermark emitted by a query's sink before it is considered removed.*

Thus, once the FPG receives the marker tuple from Q_j, t_f and its Backward Provenance have been processed, and the query can be considered removed. Consequently, all sink tuples and their contributors from Q_j have been processed.

D4.2 R2: Expiration Promise

To keep the expiration promise, a tuple may only be marked as expired once it is not being processed anymore nor potentially processed again by any query. While arbitrarily late labelling of tuples as expired may facilitate keeping the promise, it introduces latencies for downstream processing; thus, the labelling should happen as close in time as possible while fulfilling the expiration promise.

D4.2.1 Additions: $R_i = +Q_i$

For SℚP-FP, Equation 4.1 yields an upper bound on the expiration threshold for any tuple as a function of the watermarks of the sinks at time ω and of the individual aliveness constants U_Q (see Equation 4.2) of each query. We formulate here an extension:

$$T(\omega) = \min_{Q \in \mathfrak{Q}(\omega)} \left(W_Q^\omega - U_Q \right). \tag{4.3}$$

Note that, keeping track of every individual sink watermark, this is a tighter version of the bound used in [153]. Observe that, in contrast to SℚP-FP, this formula now incorporates the dynamic set $\mathfrak{Q}$, whose constitution is dependent on time ω. At time f_i, Q_i becomes part of $\mathfrak{Q}$, and at the next time instance (for infinitesimal $\epsilon > 0$)

$$\begin{aligned} T(f_i + \epsilon) &= \min_{Q \in \mathfrak{Q}(f_i) \cup \{Q_i\}} \left(W_Q^{f_i+\epsilon} - U_Q \right) \\ &= \min \left[\min_{Q \in \mathfrak{Q}(f_i)} \left(W_Q^{f_i+\epsilon} - U_Q \right), W_i^0 - U_{Q_i} \right], \end{aligned} \tag{4.4}$$

where W_i^0 is the first watermark of Q_i's sink. The immediate adoption of tuple lifetime and watermark of Q_i into the threshold ensures that no tuple expiring in Q_i can be reprocessed by another query, and the correct marking of tuples as expired *before* g_i (when Q_i actually starts processing tuples).

To ensure that Q_i itself does not reprocess tuples expired elsewhere, we introduce the *trimmed source*:

Definition D2. *A* trimmed source *(trSource) is a source of a query Q_i that only forwards tuples t with timestamp greater than a lower bound L_i to downstream operators, $t.ts > L_i$.*

Thus, for all tuples t produced by a trSource, it is true that $t.ts > L_i$. We also introduce the *expected watermark*:

Definition D3. *The first or* expected (sink) watermark $W_i^{f+\epsilon}$ *of query* Q_i *with a trSource after* f_i *has value* $W_i^0 = L_i$.

Note the first watermark of Q_i can not be known before time $g_i > f_i$, where Q_i begins processing tuples. L_i is a valid watermark as Q_i will not output sink tuples with smaller timestamps due to its trSource. We can thus replace W_i^0 in Equation 4.4 with L_i. Which L_i should be chosen for a new query?
(i) ***E-safe***: Choosing $L_i \geq T(f_i)$ for the trSource of Q_i will prevent Q_i from processing tuples expired in the queries in $\mathfrak{Q}$ before f_i, per the construction of T in Equation 4.3. Any choice of $L_i \geq T(f_i)$ is *E(xpiration)-safe*.
(ii) ***W-slack***: However, some values of $L \geq T(f_i)$ can result in inconsistencies with watermarks. The minimum watermark among all queries, at and after f_i, is

$$\begin{aligned} W_{min}^{f_i} &= \min_{Q \in \mathfrak{Q}(f_i)} W_Q^{f_i} \\ \rightarrow W_{min}^{f_i+\epsilon} &= \min\left[\min_{Q \in \mathfrak{Q}(f_i)} W_Q^{f_i+\epsilon}, L_i\right]. \end{aligned} \tag{4.5}$$

Thus, to enforce $W_{min}^{f_i} \leq W_{min}^{f_i+\epsilon}$, L_i must be larger than $W_{min}^{f_i}$ - or else the minimum watermark could decrease, conflicting the Dataflow model that assumes non-decreasing watermarks *and* interfering with modern SPEs's watermark implementations. We say any choice of the lower bound such that $T(f_i) \leq L_i < W_{min}^{f_i}$ is *W-slack*.
(iii) ***T-slack:*** If we calculate the expiration threshold directly after f_i, the expiration threshold may *decrease*:

$$\begin{aligned} & T(f_i + \epsilon) && \geq T(f_i) \\ \xrightarrow{4.4} & \min\left[\min_{Q \in \mathfrak{Q}(f_i)} (W_Q^{f_i+\epsilon} - U_Q), L_i - U_{Q_i}\right] && \geq T(f_i). \end{aligned} \tag{4.6}$$

Since the first term in the minimum on the LHS is greater or equal than the RHS (as watermarks never decrease), we must choose $L_i \geq T(f_i) + U_{Q_i}$ to enforce the threshold to not decrease. If L_i is chosen otherwise, we say it is *T-slack*.

Figure D3 (a) shows the various types of slack and their effect on the amount of input data that can potentially be ingested by a newly added query: The smaller the value of L_i, the more data query Q_i can access through its trSource S_i. This can make it desirable to choose L_i as low as possible while still being E-safe; however, this may introduce the various types of slack. Choosing L_i greater than or equal to

$$\bar{L}_i := \max\left(T(f_i) + U_{Q_i}, W_{min}^{f_i}\right) \tag{4.7}$$

thus is neither W- or T-slack.

D4.2.2 Consecutive Additions: $R_i = +Q_i$, $R_{i+1} = +Q_{i+1}$

For not only one but consecutive query additions, further difficulties with a T-slack choice of L_i arise. For successive additions R_i and R_{i+1}, where both

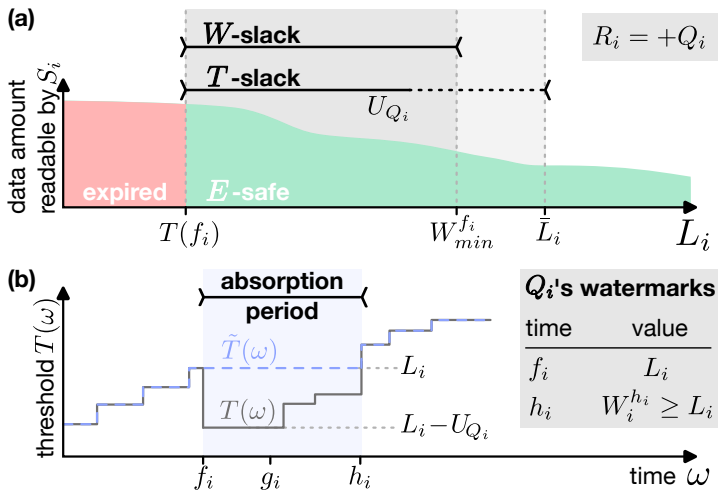


Figure D3: (a) Effects of the lower bound L_i of the trSource S_i of query Q_i. (b) Evolution of the threshold with non-decreasing enforced (light blue upper curve) for T-slack additions.

times the lower bound is chosen to equal the respective expiration threshold, the query added upon R_{i+1} may reprocess expired tuples *which is not E-safe*: For some positive time interval (f_i, h_i) after the first transition has finished, the threshold value $T(\omega)$ may be smaller than $T(f_i)$ (T-slack). Then, if the second query's addition is timed such that $f_{i+1} \in (f_i, h_i)$, the choice $L_{i+1} = T(f_{i+1}) < T(f_i)$ for the second addition will result in the query added in this transition having access to source tuples with timestamps smaller than $T(f_i)$ - and those tuples have already been marked as expired. This situation is sketched in Figure D3 (b) (dark grey curve). Thus, in this case T-slack is *not* E-safe. Then, one of two precautions is required: (i) introducing an *absorption* period after each addition, during which no further transition may begin, and which ends once the threshold has increased over its previous maximum value; or (ii) enforcing a *non-decreasing* threshold value as

$$\tilde{T}(\omega) = \max\left(T(\omega), \max_{\omega' < \omega} \tilde{T}(\omega')\right), \tag{4.8}$$

s.t. $\tilde{T}$ is always the larger of the value calculated from (4.1) and the earlier maximum of $\tilde{T}$. Note that, for (ii), the threshold will effectively keep *memory* of the tuples expired in the past. Figure D3 (b) shows both the non-decreasing choice for the threshold (light blue) as well as the absorption period.

To sum up, Q_i will not reprocess expired tuples for any L_i within the E-safe region combined with precaution (i) or (ii).

D4.2.3 Removals: $R_i = -Q_j$

When a query is removed, it can no longer ingest source tuples nor forward sink tuples. Thus, it must be excluded from considerations about the tuple

expiration threshold, which is achieved by calculating the threshold without the watermark from Q_j and the tuple lifetime U_{Q_j}. This may at most *increase* the threshold value and the minimum watermark, thus not incurring any slack. However, in the case of Q_j being the last query to be removed, such that $\mathfrak{Q} \xrightarrow{R_i=-Q_j} \emptyset$, the threshold behavior is undefined as the minimum of an empty set. Should a query then be added, it must also be hindered from ingesting tuples expired before its addition. Thus, to be *E-safe* even in this scenario, the last defined value of the threshold has to be remembered. To do this, one has to choose the threshold as Equation 4.8, such that the last value is automatically remembered[(D4)].

R1 and R2: Summary

Additions and removals are *E-safe* and complete for the following choices:

- a marker tuple as final watermark per query sink (cf. Definition D1)
- a lower bound $L_i := \bar{L}_i$ (cf. Equation 4.7) on input tuples per query source
- a non-decreasing threshold (cd. Equation 4.8)

D5 Algorithmic Implementation

Algorithm D1 RequestHelper (RH)

```
   state 𝔔, i    set of added queries & counter to assign ID to them
 1: procedure ADD(U, DAG)                ▷ U: max aliveness constant, DAG:
                                           query operator topology
 2:     sendRequest(i, U)                    ▷ send query ID and U to FPG
 3:     L ← waitReply()                      ▷ wait for reply carrying L
 4:     deploy(i, Q, L)                      ▷ deploy query to SPE
 5:     𝔔 ← 𝔔 ∪ {i},    i++
 6: procedure REMOVE(j)
 7:     if j ∈ 𝔔 then
 8:         insertMarker(j)                  ▷ insert marker to query with ID j
 9:         asyncCancel(j)                   ▷ cancel query execution asynchronously
10:         𝔔 ← 𝔔 \ {j},    i++
11:         waitReply()                      ▷ wait until FPG has received marker
```

Nona's implementation consists of two components (see Figure D2): the ForwardProvenanceGenerator (FPG), a streaming-based operator receiving a stream of backward provenance and watermarks from the sinks of the added queries, and the RequestHelper (RH). A user submitting requests interacts with the RH via the procedures ADD() and REMOVE(), which are detailed

[(D4)]When opting for an absorption period instead of a non-decreasing threshold, one would need to memorize the final valid threshold value by hand.

in Algorithm D1. The RH then interacts with the FPG via message queues: The RH uses method sendRequest() to relay a message to the FPG, and the blocking waitReply() to wait for a response message from the FPG; the FPG sends such a response message using sendReply(). Meanwhile, the FPG switches in a loop between the following: (1) it uses procedure onRequest() to check for a new message (and returns if there is none), and (2) it invokes onSinkTuple(), onWatermark(), or onMarker() (a special type of watermark, explained below) depending on the next element in its input stream (see Algorithm D2), and returns if there is no next element. The implementation provided here is for the case of additions that are neither T- nor W-slack (see Section D4.2) and uses a non-decreasing threshold (see Equation 4.8).

The following argumentations accompany the algorithmic implementations explained in pseudocode in Alg. D1,D2.

Claim D1. *DQ-FP implemented using the components from Algorithm D1 and Algorithm D2 fulfill requirements R1-R4.*

Proof. The proof is broken down into four subsections corresponding to the individual requirements.

Algorithm D2 ForwardProvenanceGenerator (FPG)

```
    state  𝕎, 𝕌              maps from query ID to watermark / U
           W_min ← −∞         minimum watermark in 𝕎
           T ← −∞             threshold
 1: procedure onRequest(i, U)          ▷ i: query ID, U: max aliveness constant
 2:     𝕌.put(i, U)
 3:     L ← getLowerBound(𝕎, 𝕌)                ▷ according to Equation 4.7
 4:     𝕎.put(i, L)                    ▷ store L as first watermark from i
 5:     sendReply(L)                                          ▷ send L to RH
 6: procedure onMarker(j)
 7:     𝕌.remove(j)                                  ▷ remove query with ID j
 8:     𝕎.remove(j)
 9:     T ← updateThreshold(𝕎, 𝕌)              ▷ according to Equation 4.8
10:     W_min ← minValue(𝕎)
11:     sendReply(j)                  ▷ send ack to RH that j was removed
12: procedure onSinkTuple(t_K)
13:     if getQueryID#(t_K) not in 𝕎 then
14:         output
15:     ID_K = getID(t_K)
16:     emit(⟨W_min, [ID_K, t_K]⟩)                          ▷ emit sink vertex
17:     sourceTuples ← getBProvenance(t_K)
18:     for t_S : sourceTuples do
19:         ID_S = getID(t_S)
20:         if (t_S.τ, ID_S) ∉ ℙ then
21:             emit(⟨W_min, [ID_S, t_S]⟩)                  ▷ emit source vertex
22:             ℙ ← ℙ ∪ {(t_S.τ, ID_S)}
23:         emit(⟨W_min, [ID_S, ID_K]⟩)                     ▷ emit edge
24:     emit(⟨W_min, [ID_K]⟩)                          ▷ emit sink vertex label
25: procedure onWatermark(W, k)                                  ▷ k: query ID
26:     if k not in 𝕎 then            ▷ skip watermarks of removed queries
27:         output
28:     𝕎.put(k, W)
29:     W_min ← minValue(𝕎)
30:     T ← updateThreshold(𝕎, 𝕌)              ▷ according to Equation 4.8
31:     for (τ, ID_S) ∈ ℙ do
32:         if τ > T then
33:             output
34:         emit(⟨W_min, [ID_S]⟩)                      ▷ emit source vertex label
35:         ℙ ← ℙ \ {(τ, ID_S)}
```

R1 - Completeness To achieve completeness, every sink tuple observed by a source as well as the contributing source tuples, the connecting edges, and respective label tuples must be part of the graph stream. As the definition of *observed* in this context requires t_K to be received by Nona, the latter must produce these elements correctly for every t_K it receives. In $\mathcal{A}$D2:12 (Algorithm D2, line 12), upon receiving a t_K, procedure onSinkTuple(t_K) is invoked, which retrieves the source tuples t_S contributing to t_K using getBProvenance(t_K) ($\mathcal{A}$D2:17). In the loop following this, the source vertices for the t_S and an edge per relation $t_S \rightsquigarrow t_K$ are produced, and finally the sink vertex and its label. Procedure onWatermark updates the internally stored list of watermarks and Us to calculate the current threshold ($\mathcal{A}$D2:30) and emits labels for sink tuples once these are expired according to the threshold. For the removal of query Q_j, all observed tuples must be processed before the query is removed from FPG's internal state. This happens through the use of the *marker* tuple ($\mathcal{A}$D1:8) during removals: the marker travels in the FPG's input stream like a watermark (cf. Definition D1) and signals that no other tuples from Q_j will be observed.

R2 - Expiration promise (i) For additions, to keep the expiration promise, the lower bound L_i of the added query must be set such that the new query Q_i cannot process expired tuples. Additions begin by invoking add() ($\mathcal{A}$D1:1), passing the DAG of the new query and the maximum aliveness constant U. The RH transmits this request to the FPG along with the U value and blocks; the FPG then uses U to calculate the current safe L_i ($\mathcal{A}$D2:3) and update its watermark map afterwards with the expected watermark ($\mathcal{A}$D2:4, see Section D4.2). L_i is sent back to the RH, which deploys the DAG with the value of L_i for its trSource ($\mathcal{A}$D1:4). As from now on Q_i's watermark is considered by the FPG and Q_i itself can only process non-expired tuples, the expiration promise is kept.

(ii) For removals (see Section D4.2.3), the threshold is updated in a non-decreasing way in $\mathcal{A}$D2:9 upon reception of the marker by the FPG. When considering a removed query for the threshold and watermark calculation, the latency in marking tuples as expired would grow indefinitely (as eventually watermark and aliveness constant of the removed query dominate the threshold value). To prevent this, onMarker() purges the removed query from its internal state before updating the threshold in a non-decreasing way.

R3 - Ordering and uniqueness The graph elements are produced by procedures onSinkTuple() and onWatermark() of the FPG depending on the next element in the latter's input stream. In line with the proof of the algorithmic implementation from [153], these procedures fulfill the requirements of Forward Provenance towards deduplication and ordering of graph elements in absence of transitions, hinging on using the current internal watermark of the FPG as the timestamp of produced graph elements and using the threshold value for deduplication and production of source vertices. As a modification to this for D$\mathbb{Q}$-FP, onSinkTuple() uses the current minimum watermark value which is calculated by onWatermark() every time a new watermark is

received from one of the added queries. As shown in Section D3, the watermark is non-decreasing and thus viable to use as a reference point for ordering tuples in the presence of transitions. Also, onWatermark() updates the threshold value on each received watermark ($\mathcal{A}$D2:30) and thus always uses a threshold value that is E-safe. As the threshold never decreases, this ensures that a unique source vertex will be produced once no further sink tuple may be produced to which the source vertex contributes. This guarantees deduplication.

R4 - Responsiveness Calling add() and remove() does not result in any operation halting the execution of running queries. In fact, while using the expected watermark in $\mathcal{A}$D2:4 may stall progression of the minimum watermark of the FPG (and thus the emission of graph elements), all added queries continue processing tuples and producing results. The minimum watermark of the FPG continues growing once the newly added query has begun producing tuples. Removals have even less effect on added queries as they at most increase the values of the threshold and the watermark (see Section D4.2.3). By stopping query execution in the SPE asynchronously ($\mathcal{A}$D1:9) and immediately proceeding, removal time is minimized. Thus, responsiveness is guaranteed during transitions.

□

D6 Evaluation

We evaluate the performance of Nona in two scenarios: (i) *static* (without transitions) - we benchmark Nona with Ananke [153], the state-of-the-art in SQ-FP, to evaluate implicit architectural overheads of Nona in Section D6.3 and the latter's impact on the performance of queries in Section D6.4. (ii) *dynamic* - we evaluate Nona's transition behavior on centralized and distributed hardware in Section D6.5, and stress-test it under high load. All experiments can be reproduced using the code at [89].

D6.1 Metrics

- *Rate*: rate (in tuples/second) at a query's source[(D5)].
- *Latency*: delay (in seconds) between a query's reception of the last source tuple contributing to a sink tuple and the observation of the latter at the query's sink.
- *Memory*: RAM (in MB) used by SPE and helper programs.
- *CPU utilization*: percentage of total CPU time utilized over all available processors.
- *Transition duration*: for additions, time differences $f_i - s_i$ and $g_i - f_i$ / for removals, $f_i - s_i$ (all in ms).

[(D5)] As described in Section D6.2, we use Apache Flink as our SPE. As Flink has flow control, the rate at the source is correlated to the throughput of the query.

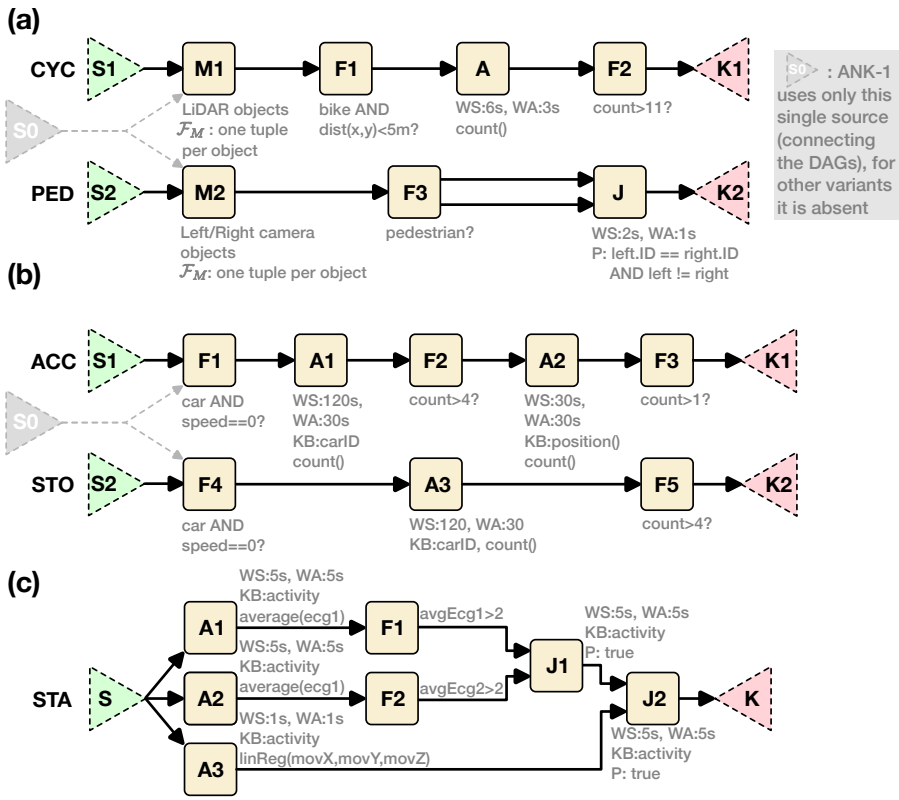


Figure D4: Queries used in the evaluation: (a) Vulnerable Road Users, (b) Linear Road, (c) Statistical Summarization.

D6.2 Setup

Hardware and Software We implement Nona using the SPE Apache Flink [29], with every query and the FPG operator running in a separate task slot. Apache Kafka [9] implements the common data source, the tuple queue between query sinks and FPG operator, and passes messages between FPG and RH (see Section D3.1). RH is implemented as a `bash` script. Our evaluation covers both low- and high-powered systems: (i) Odroid-XU4 [88], with Exynos 5422 Cortex-A15 (2Ghz) and Cortex-A7 octacore CPUs, 2GB RAM, Ubuntu 18.04.6, OpenJDK 1.8.0_252; (2) single-socket Intel Xeon-Phi server with 72 1.5GHz cores (4-way hyper-threading), 32KB L1 and 1MB L2 caches, 102GB RAM, CentOS 7.9.2009, OpenJDK 1.8.0_161. Both use Apache Kafka 3.2.1. and Flink 1.10.0 (running on the Cortex-A15s on the Odroids) We stress the execution environment in each experiment.

Queries We use five queries from three use cases:

- *Vulnerable Road Users* [153]: an object annotation use case for in-vehicle computer vision systems using data from the Argoverse Tracking

dataset [35]. *Queries* (Figure D4 (a)): **PED**, detects crossing pedestrians; **CYC**, detects cyclists.

- *Linear Road Benchmark* [12]: a performance evaluation framework to benchmark SPEs in real-time traffic analysis applications. *Queries* (Figure D4 (b)): **STO**, identifies stopped vehicles by analyzing consecutive reports of zero speed and a consistent position; **ACC**, detects accidents by examining groups of cars that have stopped at the exact same position.

- *Statistical Summarization*: an adapted version of a query from the RIoT benchmark [176], using fitness tracker data from the mHealth project [19]. *Query* (Figure D4 (c)): **STA**, calculates two heart rate averages and the 3D linear regression parameters for motion forecasting and combines them into a summary.

Procedure The input data stream to the queries (see Figure D2) is provided by a Kafka instance running on an external machine. To model an external data source, its resource usage is omitted. All experiments are repeated at least 10 times; error bars designate the 95% confidence interval[(D6)].

D6.3 Implicit Overheads

To quantify implicit overheads from Nona's architectural choices, we use as baseline the rate, latency, memory and CPU consumption of Ananke (ANK-1). Note that, since the deployed queries are known a-priori in static cases, ANK-1 applies optimizations like merging queries into a single DAG, chaining of operations (reducing tuples (de-)serialization), and reusing operators. We first compare ANK-1 with a variant ANK-mD (**m**ultiple separate **D**AGs), which omits static-only optimizations and uses an architecture identical to Nona (see Figure D2), supporting the *theoretical* deployment of new queries at runtime (while otherwise re-using the ANK-1 operator as FPG for forward provenance and giving no guarantees for D$\mathbb{Q}$-FP). We compare 10 minute runs for query set $\{Q_1\}$ (*single*) vs. $\{Q_1, Q_2\}$ (*dual*) between ANK-1 and ANK-mD.

Figure D5 (a) shows the results for $Q_{1,2}$ = ACC on the Odroid. For a single query, ANK-mD has 10% lower rate and 20% higher latency. Differences in memory and CPU usage stem from the additional use of Kafka in ANK-mD. For dual queries, the performance of both ANK-1 and ANK-md is comparable to the single-query case, but ANK-mD's CPU usage spikes to more than double that of ANK-1. This is because ANK-mD defines *two* independent sources (one per query DAG, with ACC's one performing most of the work, as most tuples get filtered by $F1$, cf. Figure D4 (b)), while ANK-1's merged DAG defines only one. Figure D5 (b) shows the results for $Q_{1,2}$ = CYC on the server. CYC is computationally heavier than ACC, and produces significantly more provenance. For a single query, ANK-1's static optimizations lead to better performance at lower resource utilization. For dual queries, though, ANK-1 achieves a lower rate at double the latency, as its single DAG structure lets

[(D6)] For stacked bar plots, the error bars are respective to the sum of the bars.

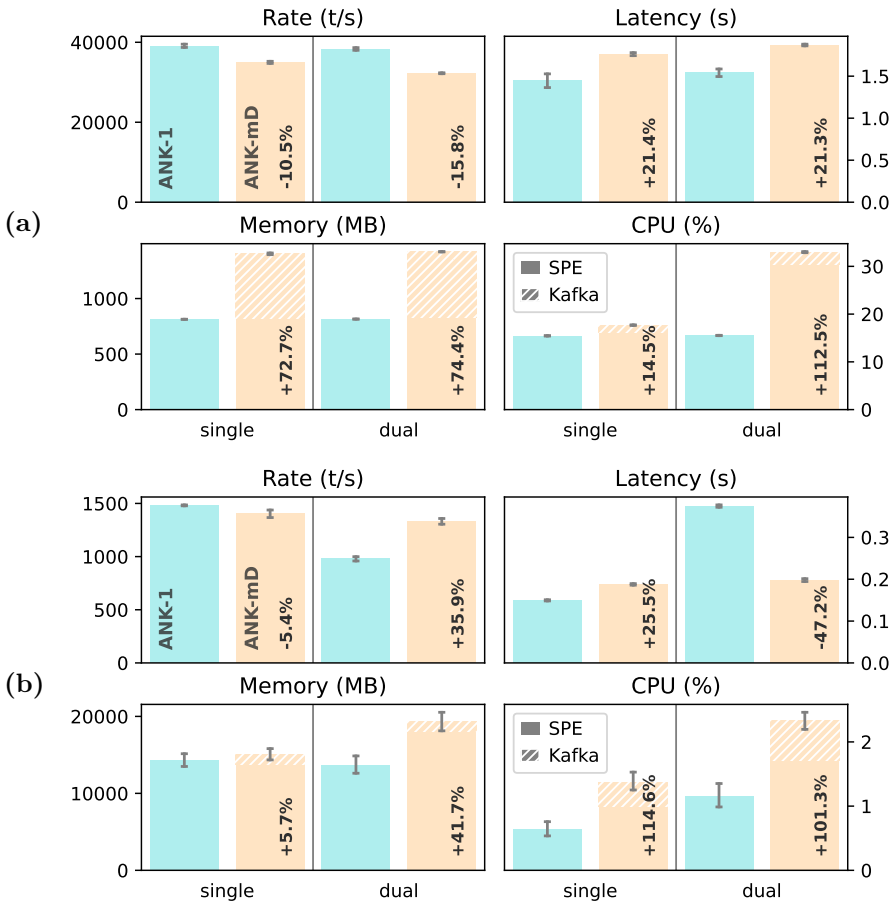


Figure D5: Implicit overheads evaluation: (a) {ACC} vs. {ACC,ACC}, (b) {CYC} vs. {CYC,CYC}.

backpressure (resulting from high provenance rates) propagate from the tail of the DAG (generating SQ-FP) to the sources. As ANK-mD has Kafka as a queue between FPG and queries, it can process at rates higher than the intake rate of the FPG.

D6.4 Performance of Nona under Static Query Sets

Having quantified architectural overheads, we now benchmark Nona (NONA) in the static case with ANK-mD. To analyze NONA in a static scenario, we add queries in the first minute of each experiment, to then remove this warm-up phase of query deployment from the evaluation. The experiments consist of executions of at least 10 minutes, for three different query sets: {ACC,STO}, {CYC,PED} and {STA}.

Figure D6 shows the results for query sets {ACC,STO} (a) and the compu-

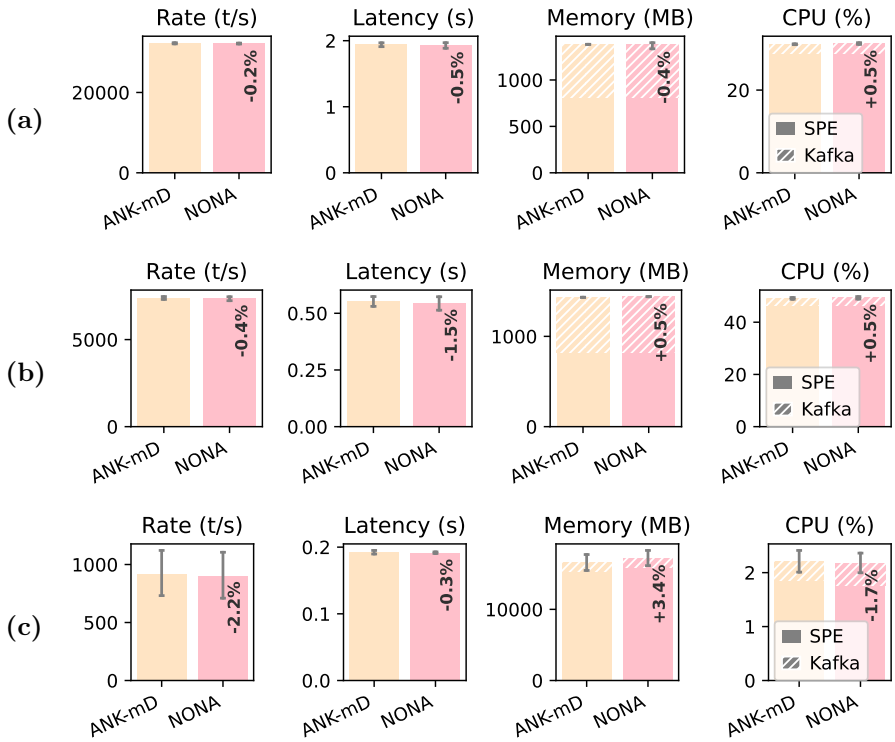


Figure D6: Static evaluation: (a) {ACC,STO}, (b) {STA}, (c) {CYC,PED}.

tationally heavy {STA} (b) on the Odroid, and {CYC,PED} (c) on the server. Evidently, the performances of ANK-mD and NONA are indistinguishable. The large error bars for the rate in (c) are because PED's source runs significantly faster than that of CYC, leading to large confidence intervals in the statistical summary over individual sources. The results indicate that any additional overheads from NONA over ANK-mD related to the former supporting DQ-FP vanish in the static case. Thus, even in dynamic scenarios, NONA's performance is almost indistinguishable from the state-of-the-art (ANK-mD, given equal architecture) during static periods.

D6.5 Performance of Nona under Dynamic Query Sets

To investigate the performance of Nona in the dynamic setting, we design an experiment procedure as follows:

$$\emptyset \xrightarrow{R_1=+Q_1} \{Q_1\} \xrightarrow{R_2=+Q_2} \{Q_1, Q_2\} \ldots \xrightarrow{\cdots} \{\ldots, Q_N\},$$

waiting 60 seconds between requests, followed by

$$\{\ldots, Q_N\} \xrightarrow{R_{N+1}=-Q_N} \{\ldots, Q_{N-1}\} \ldots \xrightarrow{R_{2N}=-Q_1} \emptyset.$$

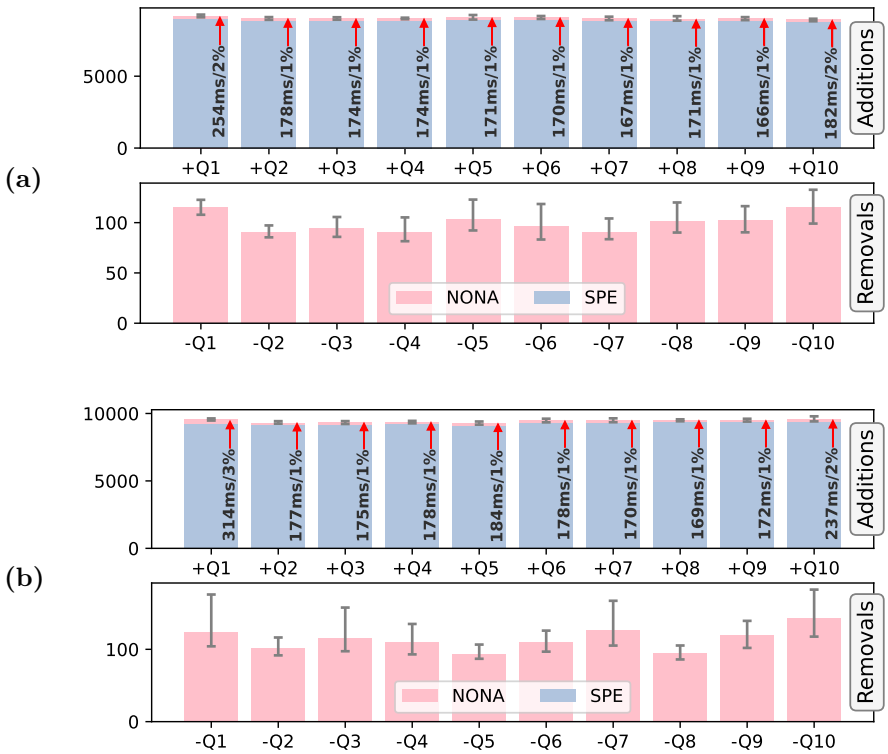


Figure D7: DQ-FP (server): Transition times [ms]. (a) ACC, (b) STA (note: share of Nona in *Additions* highlighted with arrow).

We can thus measure how additions and removals scale with an increasing number of already running queries. To remove effects from varying DAG topologies, we use $Q_k = Q_j$, $k, j = 1 \ldots, N$. The total experiment duration is 25 minutes per run.

D6.5.1 Real Queries - Server

For Q_k = ACC, Figure D7 (a) shows the average addition and removal durations on the server. Red bars represent the time spent solely by Nona's FPG and RH ($f_i - s_i$, see Section D4), while blue bars the time spent by the SPE until the query starts processing tuples ($g_i - s_i$). As additions first require a reconfiguration of Nona and then query deployment by the SPE, the duration is dominated by the latter (which (de-)serializes the query's DAG, instantiates operators, and connects source and sink to Kafka). Regarding only Nona, all except the first transition ($+Q_1$) take on average 170ms (or 1% of the transition time including the SPE, see text on bars). This indicates a near-constant transition time irrespective of the number of currently running queries. For removals, no time is spent by the SPE, as removals start and end with the

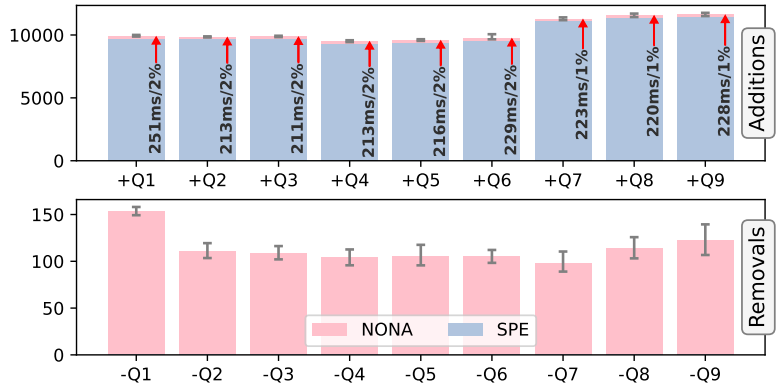


Figure D8: DQ-FP (Odroid cluster, ACC): Transition times [ms] (note: share of Nona in *Additions* highlighted with arrow).

insertion and reception of the marker tuple (see Section D5), while the query itself is stopped in the SPE asynchronously. Removals take ∼100ms. Although the marker has to propagate along the input stream to the FPG, requiring the FPG to process all preceding tuples first, removals are significantly faster than additions (ignoring even time spent by the SPE) in this experiment. Figure D7 (b) shows the transition times for Q_k = STA. Additions require on average ∼180ms (1-2% of the total addition time), almost identical to the addition times for ACC; just as removals, which take ∼110ms.

D6.5.2 Real Queries - Odroid Cluster

A single Odroid is not able to sustain the execution of many simultaneous queries due to its limited performance and number of cores. However, Nona's architecture (cf. Figure D2) allows a distributed deployment over various nodes, with one node running the FPG and RH operators, and other nodes executing one or several queries. To evaluate Nona in the dynamic scenario on Odroids, we thus deploy a *cluster* of four Odroids connected by 100MB/s ethernet. The main node is running the FPG and RH operators, and the others up to three queries each for a maximum of nine parallel queries. Figure D8 shows the results for this setup for the choice Q_k = ACC. On the cluster, additions take on average ∼225ms, while removals occur in ∼110ms. This is only 55ms (10ms) slower than on the vastly more powerful server. For the choice of Q_k = STA, additions similarly take ∼225ms and removals ∼110ms[(D7)]

D6.5.3 Synthetic Query

A long input queue of the FPG can lead to a higher latency in receiving the marker tuple. This is not visible in Section D6.5.1, where the sinks of ACC

[(D7)]The corresponding plot is omitted for brevity.

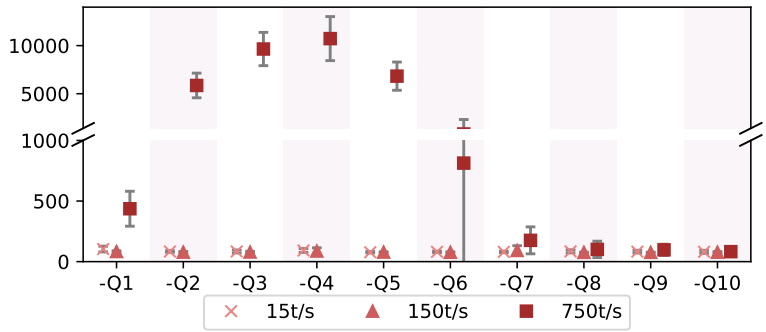


Figure D9: Dynamic evaluation (server): Removal times [ms]. 10 synthetic queries, 3 sink tuple rates (note: gapped y-axis).

(STA) forward on average 0.33 (40) sink tuples per second. To stress-test the removal mechanism for high sink rates on the server, we use a synthetic query SYN, Q_k = SYN, $k = 1 \ldots, 10$, whose sinks emit a pre-populated backward provenance graph with two source tuples per sink tuple. SYN is set to constant rates of 15t/s, 150t/s and 750t/s. Figure D9 shows the removal transition times on the server (with a gapped y-axis to accommodate all values). For 15t/s and 150t/s, there is no discernible effect, with removal times averaging circa 100ms (as in Figure D7). At 150t/s, the 10 queries have a combined output rate of 1500t/s, which is already a large number of tuples to inspect using provenance. Effects become starkly visible at 750t/s: Removal times of the first 6 queries are markedly longer, building up to 10s for Q_4's removal before decreasing again. After removing Q_6, the FPG has caught up with the incoming rate again.

Evaluation Summary

Nona is evaluated on various datasets (benchmark and real-world) and devices (server, single Odroid and Odroid cluster). As shown, its architecture may impose overheads over the state-of-the-art for light queries, while being beneficial for heavier ones. On static query sets, Nona has no further overheads (amortizing architectural differences); on dynamic ones, Nona on the server tolerates at least a combined 1500t/s before query removal times increase. Additions take constant time dominated by SPE's query deployment ($> 99.5\%$). Lastly, even a single Odroid executing the main components of Nona (FPG and RH) achieves, for additions and removals, a performance very similar to the powerful server.

D7 Related Work

Forward provenance extends backward provenance, a well-established concept in databases [38]. In stream processing, early implementations delivered only coarse-grained data relations [191], while later works provided *fine-grained* provenance, first via variable-size annotations [77] and eventually with constant per-tuple overhead [150]. [77] also presented a first investigation into tuple lifetimes. Static forward provenance (S$\mathbb{Q}$-FP) was then presented in [153], introducing both the concept as well as an implementation that requires low overheads.

To the best of our knowledge, ours is the first work extending *static* to *dynamic* forward provenance in stream processing. The notion of dynamic query sets resembles that of *ad-hoc* queries [110]. While highlighting the need for correctness and consistency, though, [110] does not focus on provenance but on resource sharing [54] and query optimization [109] between active queries. Further similarities to *elasticity* in data streaming exist [30], [169], where the query execution is dynamically optimized by, e.g., deploying new computation nodes to reduce load. Like for our work, elasticity requires low-latency reconfigurations [83] that do not halt query execution, providing continuous service without dropping tuples.

D8 Conclusions

We formalized the problem of D$\mathbb{Q}$-FP and proved it is possible to provide a forward provenance graph for dynamic and evolving sets of queries with the same guarantees offered for static sets of queries. We also presented an efficient implementation using Apache Flink and Apache Kafka and showed that its architecture incurs negligible overheads when compared to the state-of-the-art for static sets of queries. Upon changes in dynamic query sets, transitions only incur temporary sub-second latency overheads, independently of the number of running queries, up to substantially high loads.

Motivated by Figure D1's example, Nona's D$\mathbb{Q}$-FP can be used to efficiently and safely *select* data for further processing, while adapting data selection queries as knowledge is gathered or the underlying data changes. Our work can enable novel research in, e.g., provenance guarantees when live updates are applied to a query while it is processing data, as well as parallelization techniques for efficient provenance maintenance.

Chapter E

Evaluating Distributed Analysis Algorithms in VCPSs

Bastian Havers, Marina Papatriantafilou, Ashok Koppisetty, Vincenzo Gulisano

The following is an adapted version of the work published in *Proceedings of the 23rd International Middleware Conference Industrial Track, Part of Middleware 2022, p. 22-28*, as *"Ananke: A Streaming Framework for Live Forward Provenance"*. Any changes serve only to retain the consistency of this thesis.

Abstract

Highly-connected Vehicular Cyber-Physical Systems (VCPSs) offer manifold opportunities for distributing learning across the contained vehicles, road-side units and servers. However, simulating and evaluating particular distributed learning schemes poses a difficult problem in requiring realistic modeling of the vehicular fleet, communication, and the learning itself. In this work, we postulate a set of requirements for a framework simulating a complete learning workflow in a VCPS, and propose a modular architecture for it. Using a prototype implementation, we show with an example experiment the capabilities the proposed framework delivers for evaluating novel learning schemes in custom scenarios.

E1 Introduction

Using Machine Learning (ML) in Vehicular Cyber-Physical Systems (VCPSs) of smart and connected vehicles has shown manifold valuable applications like object recognition from visual data [4], vehicle maneuver planning [96], [149], or driver intention recognition [23], [166]. Applications like these for smart driver assistance functionality are projected to deliver fully autonomous driving in the near future. As the data fed to the underlying ML models is usually sensed by vehicles while the training of such models is performed centrally, vehicle manufacturers and fleet operators need to retrieve raw data from the vehicles. This can happen either via sporadic physical access to vehicles or frequent wireless transmission. While the latter is preferable to access fresh data, it nonetheless incurs variable costs for cellular broadband usage[(E1)] (this effect can be alleviated through smart preprocessing/compression on the edge [91], [153], but the issue of scaling remains). Notice that both approaches share the risk of potential exposure of sensitive user data at the central data center, which should be minimized according to privacy regulations such as GDPR. Central data gathering can thus hinder scalability of the training of models going forward due to rising transmission costs and legal requirements. Such shortcomings can be alleviated by alternatives utilizing the computational power on the system's *edge*, i.e., the vehicles' onboard devices.

The spectrum of those alternatives spans from Federated Learning (FL) [53], [57], [63], [132], where a central server aggregates only model parameters (instead of raw data) from the edge, to Gossip Learning (GL) [122], [158], [174], in which devices communicate their models directly with each other without central coordination. Comparing which of these approaches best suits the needs of a VCPS in question can be difficult due to the variable system dimensions. Leaving out the option of data collection via physical access due to its impracticality (e.g., stale data, hard to scale to large fleets), some of such dimensions include vehicular on-board capabilities, available communication channels (e.g., can vehicles employ vehicle-to-vehicle communication or only vehicle-to-cloud?), individual vehicle usage patterns dictating when vehicles are turned on and how they are moving about (influencing for example network coverage at a vehicle's location or proximity to other vehicles), the data distribution in the fleet [60] and fleet size. Taken together, these forbid a one-size-fits-all solution to decentralized learning in a VCPS.

It is costly, potentially slow, risky, and impractical to evaluate decentralized learning approaches directly in the actual fleet due to, among others, the complexity and safety requirements surrounding on-board software. Thus, a framework able to accurately simulate learning processes inside a VCPS, allowing the extraction of custom metrics that make various learning approaches comparable and parameterizable at lower costs and higher speeds, is needed

[(E1)] While cellular broadband costs have fallen sharply over the past years [199], the data amounts required to train well-performing modern ML models such as Deep Neural Networks (DNNs) over multi-dimensional data (e.g., images) are growing approximately in proportion to increasing model complexity [79], potentially offsetting decreased transmission costs per byte sent.

before turning to real-world test runs.

In this paper, we collate the requirements such a framework needs to fulfill in order to help OEMs and fleet operators in testing and evaluating practical alternatives to centralized ML, propose a general modular architecture for such a framework to guide its design, and show example results obtained with a prototype implementation that we have developed.

E2 Related Work

To the best of our knowledge, no single framework exists that answers all challenges mentioned in Section E1. Nonetheless, many complementary tools have been developed. Since we aim at discussing the challenges and needs of a comprehensive framework rather than specific implementations of it, we list and discuss some relevant related examples here.

There exist several established tools to simulate vehicular traffic, for example SUMO [131] and VISSIM [67]. Coupled with network simulators such as OMNet++ [189] or NS3 [31], these can yield combined tools to simulate and evaluate Vehicular Ad-Hoc Networks (VANETs). An overview of such combined tools can be found in [198], for example VEINS [178] or ezCar2x [171]. These tools both enable the testing of applications relying on vehicle-to-vehicle and vehicle-to-cloud communication and could serve as a basis for the framework proposed in this work. However, their starting point is the simulation of vehicles' trajectories, while fleet operators and vehicle manufacturers typically have access to unbiased real-world vehicle trajectories and may thus not require full-blown traffic simulations, which may add extra overhead. It should be noted that SUMO allows to generate routes from existing GPS data and can thus allow experimentation exclusively with real data.

Several frameworks focus solely on the aspect of learning in a distributed system, omitting any connection to VCPSs. Besides the lacking ability to implement vehicular dynamics directly, they offer no direct or only limited support for more advanced or hybrid learning strategies. Flower [24] is an open-source framework that allows to implement and experiment with various flavors of FL on actual edge devices and single-machine setups and is flexible in its support for various ML frameworks. However, its focus lies on supporting FL only; thus, support for other types of distributed learning such as GL is lacking. More straightforward tools such as FLSim [194] offer, after some customization, a stripped-down but also less feature-rich experimentation with FL strategies. The framework TensorFlow Federated [185] offers "Federated Analytics" functionality for more varied federated computations. However, for implementing strategies such as GL [93], no singular framework (comparable for example to Flower) could be found by the authors of this paper.

In [55], the authors explore and evaluate one approach to GL in a VCPS by combining the SUMO traffic and OMNet++ communication simulators with the Keras ML framework [41]. However, as that work aims at evaluating only GL, it remains open if their experimentation framework would suit the needs of a more flexible tool that allows even for hybrid solutions and simulated

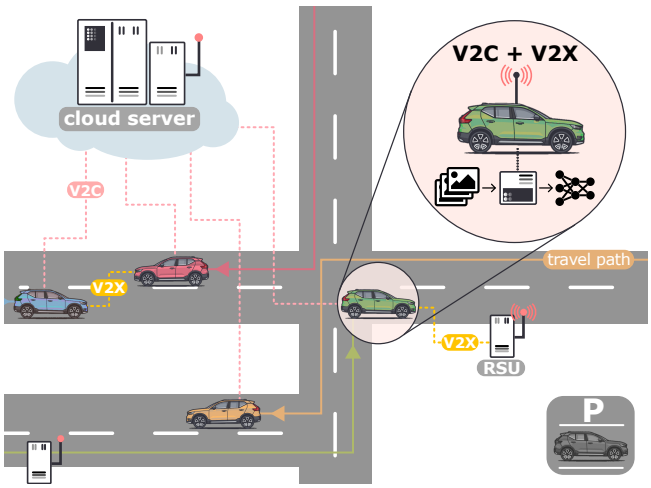


Figure E1: Sketch of a VCPS. Via V2C (vehicle-to-cloud), any car can communicate with the cloud server, while V2X (vehicle-to-anything) is local-only: between cars, and cars and road-side units (RSUs). Each car (see inset) collects data to train an ML model (generally, also cloud server and RSU are capable of training). Vehicular dynamics are dictated by their travel paths. Cars that are turned off (see: parked grey car) temporarily do not partake in the VCPS.

on-board deployment.

In summary, while all elementary parts of the sought-after framework exist, no single tool covers all required dimensions in the appropriate depth, and the complete design of such a unifying framework remains open.

E3 Problem Statement and Requirements

Offering a framework to aid in developing, evaluating, and optimizing strategies for learning from data gathered on the edge of a VCPS poses several challenges. Such a framework must be able to take the specifics of a highly evolving, connected vehicular fleet into account both on the level of the individual vehicle and the whole fleet. Furthermore, sufficiently realistic modeling of various forms of communication between various actors in the system is required, as well as modeling of the learning aspect itself. Eventually, all these aspects have to be simulated by taking into account the real-time behavior of the various components.

The learning VCPS and the contained simulated agents are sketched in Figure E1: Vehicles, equipped with an on-board unit to transform data into an ML model (see inset), a cloud server, connected to the vehicles via a V2C (vehicle-to-cloud) connection, and RSUs (Road-Side Units) that can communicate with the vehicles via short-range V2X (vehicle-to-anything) and the cloud server via a wired connection.

Preliminaries To define the requirements of the framework in detail, we introduce a few key concepts: A *learning problem* defines a real-world problem an analyst wants to solve, e.g., the predictive maintenance of a certain component of the vehicle. Solving this learning problem requires the collection of relevant data by the vehicle to learn from, with the distribution of data potentially varying strongly between vehicles. This data is then processed using *Machine Learning*, which creates and iteratively updates an ML model of the data, using techniques that span from *supervised* ones (where ground truths for each data instance are accessible, e.g., generated by humans or sensors) to *semi-supervised* or *unsupervised* ones (where no ground truth exists, for example when trying to identify anomalies or when clustering data). The learning problem is solved once one can make predictions with the ML model satisfying some requirements for the prediction's accuracy (for supervised learning, this can be measured through *testing* by for example the ratio of correct vs. wrong predictions or a prediction's closeness to a ground truth; for unsupervised learning, this could be a measure for the performance of the clustering).

How the learning problem is solved is defined through the *learning strategy*, describing how data and models are exchanged between vehicles and other actors in the system, and how and where models are trained and potentially tested. Two example learning strategies for supervised learning are the following[(E2)]:

Federated Learning The strategy proceeds in rounds. In each round, the cloud server selects a subset of vehicles and transmits to them a so-called global model. Each receiving vehicle v_i uses its local data to fine-tune (retrain) the global model locally, then sends the retrained model w_i back to the cloud server. The latter aggregates the received models into a new global model w, using for example Federated Averaging: $w = \sum_i w_i \cdot d_i / (\sum_j d_j)$, where d_k is the data amount on vehicle k (as presented in [139]).

Opportunistic $\rightsquigarrow$ Each vehicle v_i begins by training its own local model. Upon getting close in space to another vehicle v_j, both vehicles exchange their models w_i, w_j, retrain the received model, and send it back to the sender, who aggregates the received model with its original model. Thus, each vehicle plays the role of a cloud server in FL for all vehicles in its vicinity (as presented in [55]).

A major aspect of a VCPS affecting the feasibility of various learning strategies is the different modes of communication between actors. We differentiate two main types:

a) *Long-range cellular* ***Vehicle-to-Cloud (V2C)***: Vehicles can employ metered cellular connections to connect with cellphone towers, using e.g., 4G/LTE or 5G [135] communication technology. As cellphone towers are connected to the Internet, V2C allows vehicles to communicate with servers located at the vehicle manufacturers. Communication speeds achievable via V2C are the same as those achievable by mobile phones using the same cellular technology and can range from 1000 to more than 10000KB/s in ideal conditions. As

[(E2)] Note that, implicitly, it is assumed that in these approaches the ground truth of the learning problem can be generated on the vehicles themselves

sketched in Figure E1, the cloud server can, barring coverage issues stemming from e.g., tunnels, connect to any vehicle that is turned on.

b) *Short-range* ***Vehicle-to-X (V2X)***: This second family of connections is more short-ranged. It encompasses various standards such as *IEEE 802.11p* (relying on WiFi) or the more recent *Cellular-V2X* (relying on 4G/5G) to enable vehicle-to-vehicle communication (V2V) or the communication via and with road-side units (RSUs). Line-of-sight range of these connections can exceed 1000m, although this range is reduced in the presence of obstacles [163].

The viability of V2X communication is strongly dependent on the vehicles' *spatial dynamics*, i.e., how and when they move along a given network of roads to come into proximity of other vehicles and RSUs. As an example, Figure E1 shows each vehicle's travel path, dictating encounters in the VCPS.

Requirements list Using the above concepts and definitions, the framework should eventually allow the user to evaluate, for a given learning problem, spatial dynamics and modes of communication, a specific learning strategy using metrics relevant to the user.

Based on our own experiments with evaluating and refining learning strategies in a VCPS (for example as part of the Edge Lab initiative [1]), as well as reports and research from various OEMs [57], [80], we have postulated the following key requirements for the sought-after framework:

1. *Realistic fleet model.* The framework needs to be able to model the dynamics of each vehicle in a fleet, consisting of the vehicle's spatial trajectory as defined by real mobility data and its current state, which is depending on the state of the other system actors as well as external factors (e.g., a vehicle could be turned off during the system's evolution by the driver, making it unavailable).

2. *Realistic ML support.* To solve the learning problem, the actors in the system need to be able to perform the training, exchange, and testing of ML models, and do this in accordance with real-world hardware capabilities of modern vehicles. Support for various types of ML models is required to be able to tackle the manifold learning problems arising in VCPSs, and to handle various data distributions and preprocessing steps that the learning problem dictates.

3. *Realistic communication model.* Taking into account the two forms of communication laid out previously, the framework has to realistically model how vehicles communicate with each other, with RSUs, and with the central server (the specifics of this are defined by the VCPS at hand and by the intended learning strategy). Depending on the state and the location of actors, communication may or may not be possible at a given point in time, and may fail at any time.

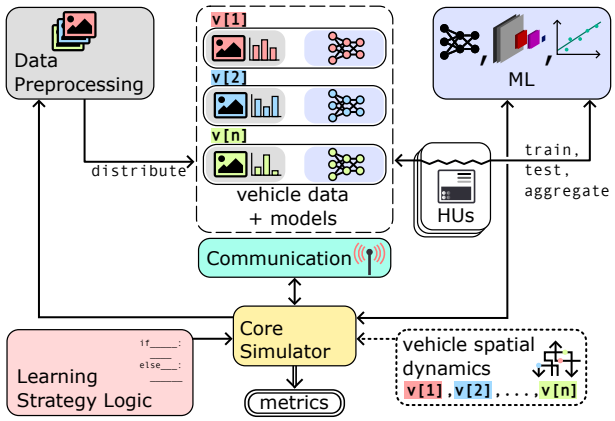


Figure E2: Proposal of the framework's modular architecture, consisting of various modules (colored boxes, see Section E4) centered around a Core Simulator (HU: Hardware Unit; v[i]: i-th agent).

4. *Fine-grained metrics.* The set of supported metrics should encompass the accuracy of the ML models in the system at various points in time and the volume of communication transmitted via the various communication channels, to enable a thorough evaluation of various learning strategies. The implementation of more custom metrics should be possible, such as computational workloads of individual vehicles or the provenance of data.

5. *Flexible learning strategy support.* The framework should allow the flexible implementation and parametrization of learning strategies to allow for easy experimentation and iteration. This means supporting centralized ML, FL, GL, as well as hybrid approaches.

6. *Quick execution.* The framework should realize a significant speed-up over an experiment in a real VCPS and reduce unnecessary overheads to allow quick experiment repetition when varying learning strategies.

With these requirements defined, we will in the following section present a proposal for a framework architecture to tackle the ensuing challenges.

E4 Architecture Proposal

The architecture proposed here is built around a Core Simulator, providing the elementary functionality of creating virtual agents and then proceeding in discrete steps through the simulation time, separated from the modules relevant to the learning problem to increase flexibility and usability. Furthermore, modules relevant to the learning problem may be designed as targeting the hardware platform of real connected vehicles, while the Core Simulator can be executed on off-the-shelf hardware. Figure E2 details the architecture proposal: At the center sits the aforementioned *Core Simulator* that orchestrates the remaining

modules, starting with the *Data Preprocessing* module. It provides the data residing on each of the n simulated agents, here designated $v[1], v[2], \ldots, v[n]$ (these agents are vehicles, road-side units, or the cloud server, see Figure E1). As an example, for the problem of recognizing road signs from traffic scenes, this module could crop and resize a given input data set of images of traffic scenes (e.g., recorded by actual vehicles), split the dataset into n subsets according to a predefined distribution, and assign each subset to a simulated vehicle as well as a test set to the simulated cloud server, all according to the specification of the learning problem at hand.

The *ML module*, likewise, keeps tabs on the current model(s) of each agent in the system (not all actors may have their own model), and provides functionality to train and test any model with any data and to aggregate models into new ones and assign these to certain agents. For example, in the aggregation step of Federated Learning, the ML module may read the models of a subset of vehicles, perform a weighted average of these (e.g., by the data amount on each vehicle in the subset [139]), and assign the result as the new model of the cloud server. The ML module deploys these operations to one or more *HU* (Hardware Unit), instances of the actual hardware existing within vehicles that allows achieving realistic performance and training times (while an agent is busy training, it may not be available for other operations). When not impacting performance, the HUs can run multiple operations in parallel to speed up the simulation, e.g., simultaneously training multiple agents' models. Note that the HU corresponds to the training-capable simulated agent, e.g., a vehicular on-board unit (OBU) or server hardware. Additionally, the ML module exposes metrics about the accuracy of various models in the simulated system.

Communication between agents is handled in the core simulator by a *Communication* module, which for example could be a communication simulator in itself (as in [178]). This module models the transmission of various types of data in the system per the communication type's properties, impacting the bandwidth and range of communication (for V2X, the range can be quite limited and highly dependent on the involved vehicle's position, requiring the core simulator to pass trajectory data to the Communication module). The Communication module also keeps track of the data volumes transmitted and exposes this metric to the Core Simulator. In the aforementioned FL aggregation step, the Communication module would simulate cellular transmission of the models between involved vehicles and the cloud server, covering both the transmission duration and its potential (partial) failure for vehicles that are unreachable or turning off.

Vehicle spatial dynamics enter the Core Simulator statically, e.g., as a file of GPS traces of all moving agents in the system. This supports the use of historic GPS data, but also of simulated data (pre-calculated with e.g., SUMO). As the act of learning in the VCPS is assumed to not influence individual vehicles' trajectories, it is sufficient that the spatial dynamics data of the VCPS is replayed by the core simulator.

The interaction of all modules of the system is parameterized by a set of rules given in the *Learning Strategy Logic* module, defining how the agents react

in which situation and thus encoding the learning strategy that is to be tested in a certain experiment run. A comprehensive example of a learning strategy, together with an evaluating experiment, can be found in section Section E5.2.

Eventually, the Core Simulator outputs an experiment run's metrics timestamped in simulated time to enable analysis of the system's evolution under a learning strategy.

E5 A Prototype Implementation: Roadrunner

E5.1 Implementation Details

Having presented a proposal for the architecture of the desired framework in Section E4, we will in this section present a prototype implementation, Roadrunner, that we have used for initial experiments with various learning strategies.

Roadrunner's Core Simulator is written in Java and based on a messaging scheme between simulated agents. The Core Simulator uses user-defined network speeds for the two communication types, V2C and V2X. Message transmission fails if agents are not in the appropriate state (e.g., V2X messages can only be exchanged if the participants are within range of each other, and a vehicle shutting off will result in any incoming or outgoing message failing). Vehicles' spatial trajectories are read from an input file, and at each point in simulated time, the Core Simulator will change the state of participating agents according to their current position and state (i.e., once they have been turned on or off).

The Data Preprocessing and ML modules are written in Python, based on the ML framework PyTorch [157]. The latter modules were inspired by the work of [194], and further modularized, extended, and converted into individual scripts that operate on vehicle data and models stored as files on disk. To interface with the Data Preprocessing and ML modules, the Java Core Simulator calls appropriate scripts. These perform, for example, the initial distribution of data onto the agents by splitting an original dataset into subsets and storing each subset, assigned to a particular agent, on disk, or the training of some agent's model by reading its data and model file and performing the training operation. The scripts time the duration of their execution and pass this value to the Core Simulator to appropriately model the time spent by agents in various states. The ML computations themselves are executed on a GPU as an OBU stand-in (as GPUs are expected standard hardware on smart vehicles) using build-in functionality of PyTorch[(E3)].

Using the logging tool Log4j, metrics are continuously extracted from the simulation to represent the state of every actor, the accuracy of the ML models in the simulation, and data transmission volumes at every point in simulated time.

[(E3)] See `https://pytorch.org/docs/stable/cuda.html`

E5.2 Sample Experiment: Testing an Opportunistic Learning Strategy

To exemplify the understanding our framework can enable, we show an experiment from a real-world example. As shown in early works on FL [139] (see Section E3 for a primer on the FL learning strategy), increasing the number of participants in an FL round can be one way to increase the accuracy of the final model. However, when deploying FL in a VCPS, and connecting cloud server and vehicles with a V2C connection, contacting additional vehicles per round results in increased cellular costs. Inspired by Opportunistic Learning (see Section E3), we explore the addition of V2X to increase the number of vehicles reached in each round. FL uses Federated Averaging (FA, see Section E3), which is mathematically associative, to aggregate a new model through *intermediate aggregation* (see Figure E3). Around this idea, we designed the learning strategy `OPP` (opportunistic):

> **Server**: Send latest global model w to R random vehicles ("reporters") via V2C, start round timer. At end of round, request new models from reporters. Aggregate received models into new global model via FA, then start next round.
> **Reporters**: Upon receiving w from server, retrain w. Upon opportunistically meeting a non-reporter vehicle, send to it w via V2X. Wait to receive back retrained model and aggregate it with own model via FA, to replace the own model. At end of round, send own model back to the server.
> **Non-reporters**: Upon receiving w via V2X from nearby reporter, retrain w. Send it back to reporter after retraining, if reporter is still in range. Else, discard w.

Thus, when during some round each of the R reporters aggregates the models from on average NR non-reporters, this learning strategy results in $N = R \cdot (NR + 1)$ model contributions in that round, but requires only $R \leq N$ connections via V2C[(E4)].

Figure E3 presents a sketch of a single round of `OPP` for two reporters: The circled numbers in the figure indicate the order of events (0-7) and black right-arrows indicate retraining. d_i are the data amounts used for retraining on each vehicle. In detail: (1) the model w is sent out to the reporters v_1, v_3, which retrain the model using their local data (2). (3) meeting non-reporter v_2, v_1 forwards the model w there (via V2X), where it is retrained to w_2 (4) and sent back together (via V2X) with the training data amount d_2 (5). Then (6), v_1 transmits the intermediate aggregate of its own and the model from v_2 to the server, and v_3 does the same. Finally (7), this intermediate aggregation yields a new model w' that is identical to the case in which all three involved vehicles are reporters. Intuitively, this approach should improve performance, given that the reporters get close to other vehicles during their trajectory for

[(E4)] In the worst case, no reporter will meet another vehicle for long enough to facilitate a successful exchange of weights, thus $NR = 0$ and $R = N$. Note that we disregard the case of reporters turning off during a round in this inequality, as that would also impact standard FL.

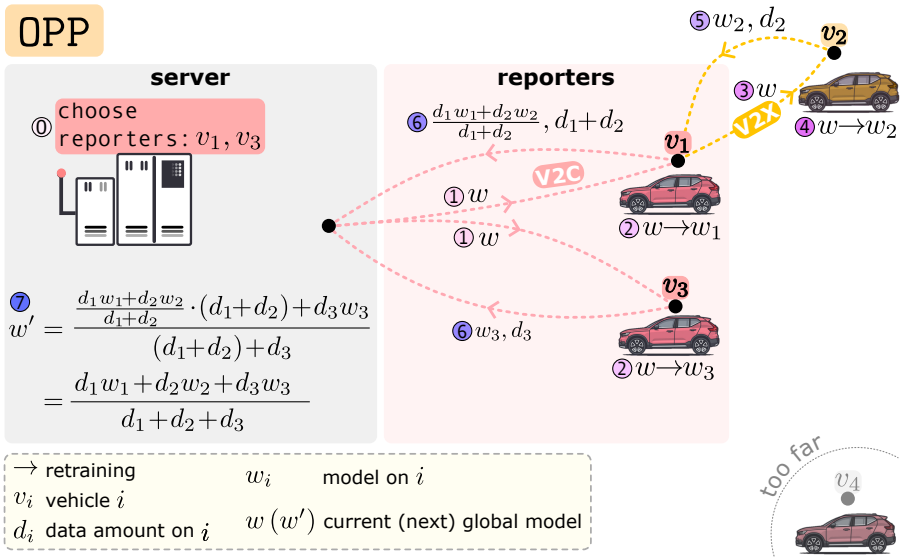


Figure E3: A single round of OPP with two reporters (v_1, v_3) and one non-reporter (v_3). Encircled numbers indicate the order of events. v_4 is too far from the reporters to participate.

times long enough to facilitate the exchange (e.g., in the sketch, v_4 is too far away from any reporter), making this approach highly dependent on the density of vehicles. Furthermore, it is intuitive that a longer round duration will give more opportunities for local aggregation of weights. Simultaneously, it will also increase the duration of the whole learning process, and increase the probability that a reporter vehicle is turned off by the driver before a round ends, effectively discarding the models collected by this reporter.

Experiment Setup To test whether these intuitions are correct, we use Roadrunner. In the following experiment, we assume a fixed V2C communication budget dictating the number of learning rounds we can perform. As a baseline case BASE, we perform FL in the VCPS, contacting 5 vehicles each round over 75 rounds of 30 seconds duration. In OPP, we also designate 5 reporters per round for 75 rounds (thus using the same V2C communication budget); however, we let reporters try to exchange their weights with encountered vehicles, and set the round duration to 200 seconds. Vehicle spatial dynamics are dictated by a proprietary real-world GPS dataset of the city of Gothenburg, Sweden. As a supervised learning problem, we choose the widely used training of a Convolutional Neural Network (CNN) for image recognition over the CIFAR-10 dataset [114] as a representative of an automotive image recognition problem. This dataset contains 60000 32x32 pixel color images in 10 classes, of which 50000 are used for training and 10000 for testing the learning algorithm. The CNN has two convolutional layers with max pooling

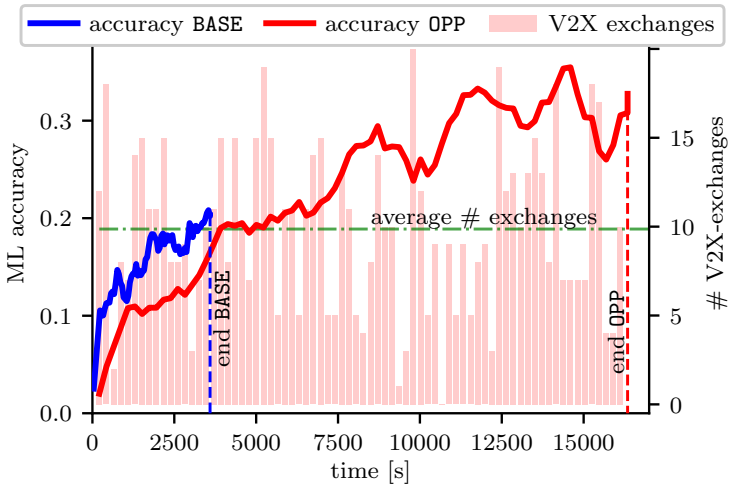


Figure E4: Evaluation experiment of learning strategy `OPP` using Roadrunner: The blue and red solid curves show the accuracy of the global model of `BASE` and `OPP`, respectively, and the bar plot shows the number of V2X exchanges that occur during a given round of `OPP`. Additionally indicated in the figure are the average number of V2X exchanges and the points in time at which 75 rounds of `BASE` and `OPP` have been completed (and the respective run ends).

followed by three fully connected layers; during training, vehicles perform two epochs of stochastic gradient descent with momentum. For the distribution of data over the vehicles, we choose a highly skewed distribution of classes in which every vehicle holds 80 samples to emulate the real-world scenario of highly personalized data. V2X range is set to 200m as an average for urban driving.

The evaluation took place on server hardware with an Intel Xeon E5-2620 v3 2.40GHz processor running the Core Simulator and an Nvidia GeForce GTX 1080 Ti graphics card as a vehicular on-board unit stand-in[(E5)].

Evaluation In Figure E4, we visualize the results from one experiment run: `BASE` (solid blue curve) finishes 75 rounds of training after 3592 seconds, while `OPP` requires 16342 seconds. The large speedup of the baseline is here explained by the much shorter round duration: as vehicles are not instructed to communicate with other vehicles in `BASE`, the round time can be set to a value that only covers the time period for transmitting and locally re-training the model. The bar plot indicates how many model exchanges via V2X occur in `OPP`, ranging from zero to 20. Thus, unlike in vanilla FL, the number of contributions to the model is not static, but varies over rounds, depending on the spatial dynamics of the vehicle (the dynamics allow or disallow encounters

[(E5)]While more modern GPUs, especially for the vehicular domain, outperform the GPU used in this experiment, it can be assumed that the available headroom for ML training in a vehicular setting is limited as on older GPUs.

close enough and long enough to exchange models via V2X). On average, just below 10 additional model contributions are thus collected per round (indicated by the dash-dotted horizontal line in Figure E4). Finally, this results in a 50% increase in final accuracy of `OPP` over `BASE`, while incurring a 4.5 times longer total real-world run time, while employing the same V2C communication budget and thus equal costs (assuming on-board training and V2X usage costs are negligible).

The ability exemplified here, of quantifying trade-offs between metrics such as data volumes, accuracy and duration, is crucial for an analyst to make informed decisions about a learning strategy and is the core contribution of any framework abiding by the requirements from Section E3.

E6 Conclusions

In this paper, we have motivated the need for a tool to evaluate various learning strategies in a Vehicular Cyber-Physical System, to help fleet operators and OEMs learn from data generated on the vehicles themselves in a fashion that is optimal for their particular fleet and use case. Furthermore, we have collated a list of specific requirements for such a tool from an (industrial) user's perspective. As we show in the Related Work, existing works partially deliver the required functionality, but no single tool fulfills the complete requirements list. Therefore, we propose an architecture for a complete tool that can evaluate various learning strategies, and we present our prototype implementation of the tool along with preliminary results to demonstrate the tool's validity. As exemplified in Section E5.2, the framework enables further understanding of the tradeoffs between dimensions such as time, cost, and accuracy when developing efficient learning strategies in a specific VCPS. In future work, we want to open-source the prototype implementation presented in Section E5 to open it for contributions from the community, with the goal of increasing its resilience and enabling thorough testing of the framework. Possible next steps are implementing additional functionality in the prototype framework, where for example the simulation of various forms of communication could be enriched by integrating existing third-party network simulators, and increasing the parallelism of the simulation to speed up learning strategy development iterations.

Bibliography

[1] AISweden, *Edge Learning Lab*, `https://www.ai.se/en/data-factory/edge-lab`, 2022 (cit. on p. 180).

[2] T. Akidau, A. Balikov, K. Bekiroğlu, *et al.*, "MillWheel: Fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013 (cit. on p. 114).

[3] T. Akidau, R. Bradshaw, C. Chambers, *et al.*, "The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *VLDB*, vol. 8, no. 12, pp. 1792–1803, 2015, ISSN: 21508097 (cit. on pp. 20, 114, 116, 146, 148).

[4] F. Alam, R. Mehmood, and I. Katib, "D2TFRS: An object recognition method for autonomous vehicles based on rgb and spatial values of pixels," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*, Springer, 2017, pp. 155–168 (cit. on p. 176).

[5] G. M. N. Ali, E. Chan, and W. Li, "Supporting real-time multiple data items query in multi-RSU vehicular ad hoc networks (VANETs)," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2127–2142, 2013 (cit. on pp. 24, 107).

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Why let resources idle? Aggressive cloning of jobs with Dolly," in *USENIX HotCloud*, 2012 (cit. on p. 108).

[7] Apache, *Beam*, 2020. [Online]. Available: `https://beam.apache.org/` (cit. on pp. 116, 117, 148, 149).

[8] Apache, *Heron*, 2020. [Online]. Available: `https://heron.incubator.apache.org/` (cit. on pp. 114, 116, 146, 148).

[9] Apache, *Kafka*, 2023. [Online]. Available: `http://kafka.apache.org/` (cit. on pp. 148, 164).

[10] Apache, *Storm*, 2020. [Online]. Available: `http://storm.apache.org/` (cit. on pp. 12, 13, 114, 116, 117, 146, 148, 149).

[11] U. S. C. of Appeals, *Intelligent transportation society of america and american association of state highway and transportation officials v. federal communications commission and united states of america*, 2022. [Online]. Available: `https://www.acea.auto/files/Platooning_roadmap.pdf` (cit. on p. 8).

[12] A. Arasu, M. Cherniack, E. Galvez, *et al.*, "Linear Road: A stream data management benchmark," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04, Toronto, Canada: VLDB Endowment, 2004, pp. 480–491 (cit. on pp. 42, 132, 165).

[13] E. A. M. Association, *What is truck platooning?* 2017. [Online]. Available: `https://www.acea.auto/files/Platooning_roadmap.pdf` (cit. on p. 7).

[14] G. V. Attigeri, M. P. MM, R. M. Pai, and A. Nayak, "Stock market prediction: A big data approach," in *TENCON 2015-2015 IEEE Region 10 Conference*, IEEE, 2015, pp. 1–5 (cit. on p. 3).

[15] J. Azar, A. Makhoul, M. Barhamgi, and R. Couturier, "An energy efficient IoT data compression approach for edge machine learning," *Future Generation Computer Systems*, vol. 96, pp. 168–175, 2019 (cit. on p. 13).

[16] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proceedings. 20th International Conference on Data Engineering*, IEEE, 2004, pp. 350–361 (cit. on p. 42).

[17] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03, New York, NY, USA: ACM, 2003, pp. 253–264 (cit. on p. 42).

[18] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05, New York, NY, USA: ACM, 2005, pp. 13–24 (cit. on p. 42).

[19] O. Banos, R. Garcia, and A. Saez, *MHEALTH Dataset*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5TW22, 2014 (cit. on p. 165).

[20] L. Battle, D. Fisher, R. DeLine, M. Barnett, B. Chandramouli, and J. Goldstein, "Making sense of temporal queries with interactive visualization," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems - CHI '16*, Santa Clara, California, USA: ACM Press, 2016, pp. 5433–5443 (cit. on pp. 114, 115, 140).

[21] P. Bedi and V. Jindal, "Use of Big Data technology in Vehicular Ad-hoc Networks," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, IEEE, 2014, pp. 1677–1683 (cit. on p. 6).

[22] E. Berlin and K. Van Laerhoven, "An on-line piecewise linear approximation technique for wireless sensor networks," in *IEEE Local Computer Network Conference*, 2010, pp. 905–912 (cit. on pp. 42, 48, 70).

[23] H. Berndt, J. Emmert, and K. Dietmayer, "Continuous driver intention recognition with hidden markov models," in *2008 11th International IEEE Conference on Intelligent Transportation Systems*, IEEE, 2008, pp. 1189–1194 (cit. on p. 176).

[24] D. J. Beutel, T. Topal, A. Mathur, *et al.*, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020 (cit. on p. 177).

[25] K. Bonawitz, V. Ivanov, B. Kreuter, *et al.*, "Practical secure aggregation for federated learning on user-held data," *arXiv preprint arXiv:1611.04482*, 2016 (cit. on p. 19).

[26] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020 (cit. on p. 3).

[27] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh, "Beyond one billion time series: Indexing and mining very large time series collections with *i*SAX2+," *Knowledge and information systems*, vol. 39, no. 1, pp. 123–151, 2014 (cit. on p. 70).

[28] L. Carafoli, F. Mandreoli, R. Martoglia, and W. Penzo, "Evaluation of Data Reduction Techniques for Vehicle to Infrastructure Communication Saving Purposes," in *Proceedings of the International Database Engineering & Applications Symposium*, ACM, 2012, 61–70 (cit. on p. 9).

[29] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015 (cit. on pp. 12, 13, 21, 28, 42, 114, 116–118, 130, 146, 148, 149, 164).

[30] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo, "Runtime adaptation of data stream processing systems: The state of the art," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022 (cit. on p. 171).

[31] G. Carneiro, "NS-3: Network simulator 3," in *UTM Lab Meeting April*, vol. 20, 2010, pp. 4–5 (cit. on pp. 26, 177).

[32] V. Cars, *Connected safety*, 2019. [Online]. Available: `https://www.media.volvocars.com/global/en-gb/media/pressreleases/251381/volvo-models-across-europe-to-warn-each-other-of-slippery-roads-and-hazards` (cit. on p. 7).

[33] K.-P. Chan and A. W.-C. Fu, "Efficient time series matching by wavelets," in *Proceedings of the IEEE International Conference on Data Engineering*, IEEE, 1999, pp. 126–133 (cit. on p. 14).

[34] B. Chandramouli, J. Goldstein, M. Barnett, *et al.*, "Trill : A high-performance incremental query processor for diverse analytics," *VLDB - Very Large Data Bases*, vol. 8, no. 4, pp. 401–412, 2015 (cit. on p. 140).

[35] M.-F. Chang, J. Lambert, P. Sangkloy, *et al.*, "Argoverse: 3D tracking and forecasting with rich maps," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8740–8749 (cit. on pp. 134, 165).

[36] Y. Chen, "Near-optimal adaptive information acquisition: Theory and applications," Ph.D. dissertation, ETH Zurich, 2017 (cit. on p. 107).

[37] Y. Chen, J.-M. Renders, M. H. Chehreghani, and A. Krause, "Efficient online learning for optimizing value of information: Theory and application to interactive troubleshooting," *arXiv preprint arXiv:1703.05452*, 2017 (cit. on p. 107).

[38] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2007 (cit. on pp. 114, 115, 140, 171).

[39] C.-M. Cheng and S.-L. Tsao, "Adaptive lookup protocol for two-tier VANET/P2P information retrieval services," *IEEE Trans. on Vehicular Technology*, vol. 64, no. 3, pp. 1051–1064, 2015 (cit. on p. 107).

[40] N. Cheng, F. Lyu, J. Chen, *et al.*, "Big Data Driven Vehicular Networks," *IEEE Network*, vol. 32, no. 6, pp. 160–167, 2018 (cit. on p. 9).

[41] F. Chollet, "Deep learning with Python and Keras: The practical guide from the developer of the Keras library," *MITP-Verlags GmbH & Co. KG, Bonn*, 2018 (cit. on p. 177).

[42] D. Christin, A. Reinhardt, S. S. Kanhere, and M. Hollick, "A survey on privacy in mobile participatory sensing applications," *Journal of systems and software*, vol. 84, no. 11, pp. 1928–1946, 2011 (cit. on p. 108).

[43] Commsignia, *Whitepaper: How is V2X viewed by consumers?* 2023. [Online]. Available: `https://www.commsignia.com/whitepaper/how-is-v2x-viewed-by-consumers/#form-info` (cit. on p. 8).

[44] R. Coppola and M. Morisio, "Connected car: Technologies, issues, future trends," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, p. 46, 2016 (cit. on pp. 6, 7, 40, 76).

[45] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012 (cit. on p. 25).

[46] S. Costache, V. Gulisano, and M. Papatriantafilou, "Understanding the data-processing challenges in intelligent vehicular systems," in *2016 IEEE Intelligent Vehicles Symp.*, IEEE, 2016, pp. 611–618 (cit. on pp. 13, 42, 76).

[47] D. Crawl, J. Wang, and I. Altintas, "Provenance for MapReduce-based data-intensive workflows," in *Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science*, ser. WORKS '11, New York, NY, USA: Association for Computing Machinery, 2011, 21–30, ISBN: 9781450311007 (cit. on pp. 114, 115).

[48] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," en, *ACM Transactions on Database Systems*, vol. 25, no. 2, pp. 179–227, Jun. 2000 (cit. on p. 140).

[49] M. Datar and R. Motwani, "The sliding-window computation model and results," in *Data Streams*, Springer, 2007, pp. 149–167 (cit. on p. 42).

[50] W. De Pauw, M. Leţia, B. Gedik, *et al.*, "Visual debugging for stream processing applications," en, in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, *et al.*, Eds., vol. 6418, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 18–35 (cit. on pp. 114, 140).

[51] T. Delot, N. Mitton, S. Ilarri, and T. Hien, "Decentralized pull-based information gathering in vehicular networks using GeoVanet," in *12th IEEE Int'l Conf. on Mobile Data Management*, IEEE, vol. 1, 2011, pp. 174–183 (cit. on pp. 24, 107).

[52] L. P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*, RFC 1951, May 1996. DOI: 10.17487/RFC1951. [Online]. Available: https://www.rfc-editor.org/info/rfc1951 (cit. on p. 14).

[53] D. Deveaux, T. Higuchi, S. Uçar, C.-H. Wang, J. Härri, and O. Altintas, "On the orchestration of federated learning through vehicular knowledge networking," in *2020 IEEE Vehicular Networking Conference (VNC)*, IEEE, 2020, pp. 1–8 (cit. on pp. 23, 24, 76, 107, 176).

[54] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering," *ACM Trans. Database Syst.*, vol. 28, no. 4, 467–516, 2003, ISSN: 0362-5915 (cit. on p. 171).

[55] M. A. Dinani, A. Holzer, H. Nguyen, M. A. Marsan, and G. Rizzo, "Gossip learning of personalized models for vehicle trajectory prediction," in *2021 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, IEEE, 2021, pp. 1–7 (cit. on pp. 177, 179).

[56] K. N. Dominiak and A. R. Kristensen, "Prioritizing alarms from sensor-based detection models in livestock production - a review on model performance and alarm reducing methods," *Computers and Electronics in Agriculture*, vol. 133, pp. 46 –67, 2017 (cit. on p. 114).

[57] S. Doomra, N. Kohli, and S. Athavale, "Turn signal prediction: A federated learning case study," *arXiv preprint arXiv:2012.12401*, 2020 (cit. on pp. 176, 180).

[58] R. Duvignau, V. Gulisano, M. Papatriantafilou, and V. Savic, "Streaming piecewise linear approximation for efficient data management in edge computing," in *34th ACM/SIGAPP Symposium On Applied Computing SAC'19*, (Limassol, Cyprus), 2019, pp. 593–596 (cit. on pp. 14, 43, 48, 66, 76, 107, 114).

[59] R. Duvignau, B. Havers, V. Gulisano, and M. Papatriantafilou, "Querying large vehicular networks: How to balance on-board workload and queries response time?" In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE, 2019, pp. 2604–2611 (cit. on pp. 40, 109).

[60] R. Duvignau, B. Havers, V. Gulisano, and M. Papatriantafilou, "Time- and computation-efficient data localization at vehicular networks' edge," *IEEE Access*, vol. 9, pp. 137 714–137 732, 2021 (cit. on p. 176).

[61] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm, "A time-series compression technique and its application to the smart grid," *The VLDB Journal*, vol. 24, no. 2, pp. 193–218, 2015 (cit. on p. 14).

[62] B. Eisenberg, "On the expectation of the maximum of IID geometric random variables," *Statistics & Probability Letters*, vol. 78, no. 2, pp. 135–143, 2008 (cit. on p. 85).

[63] A. M. Elbir, B. Soner, and S. Coleri, "Federated learning in vehicular networks," *arXiv preprint arXiv:2006.01412*, 2020 (cit. on pp. 76, 176).

[64] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet, W. G. Aref, and W. Zwaenepoel, "Online piece-wise linear approximation of numerical streams with precision guarantees," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 145–156, 2009 (cit. on pp. 42, 48, 70).

[65] M. A. Eriksen, "Trickle: A userland bandwidth shaper for UNIX-like systems.," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 61–70 (cit. on p. 54).

[66] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Kdd*, 1996, pp. 226–231 (cit. on pp. 50, 69).

[67] M. Fellendorf and P. Vortisch, "Microscopic traffic flow simulator VISSIM," in *Fundamentals of traffic simulation*, Springer, 2010, pp. 63–93 (cit. on pp. 26, 177).

[68] D. Feng, C. Haase-Schütz, L. Rosenbaum, *et al.*, "Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 3, pp. 1341–1360, 2020 (cit. on p. 3).

[69] W. J. Fleming, "New Automotive Sensors — a review," *IEEE Sensors Journal*, vol. 8, no. 11, pp. 1900–1921, 2008 (cit. on p. 6).

[70] W. J. Fleming, "Overview of automotive sensors," *IEEE Sensors Journal*, vol. 1, no. 4, pp. 296–308, 2001 (cit. on p. 6).

[71] J. H. Gawron, G. A. Keoleian, R. D. De Kleine, T. J. Wallington, and H. C. Kim, "Life cycle assessment of connected and automated vehicles: sensing and computing subsystem and vehicle level effects," *Environmental Science & Technology*, vol. 52, no. 5, pp. 3249–3256, 2018 (cit. on p. 8).

[72] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The system S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, New York, NY, USA: Association for Computing Machinery, Jun. 2008, pp. 1123–1134 (cit. on p. 140).

[73] M. Gerla and L. Kleinrock, "Vehicular networks and the future of the mobile internet," *Computer Networks*, vol. 55, no. 2, pp. 457–469, 2011 (cit. on pp. 7, 76).

[74] A. Gidenstam, B. Koldehofe, M. Papatriantafilou, and P. Tsigas, "Scalable group communication supporting configurable levels of consistency," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 5, pp. 649–671, 2013 (cit. on p. 109).

[75] GitHub, *Ananke implementation*, 2020. [Online]. Available: `https://github.com/dmpalyvos/ananke` (cit. on pp. 116, 141).

[76] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul, "Efficient stream provenance via operator instrumentation," *ACM Trans. Internet Technol.*, vol. 14, no. 1, Aug. 2014 (cit. on pp. 114, 115, 119, 141, 151).

[77] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul, "Ariadne: Managing fine-grained provenance on data streams," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '13, New York, NY, USA: Association for Computing Machinery, 2013, 39–50 (cit. on pp. 16, 119, 151, 171).

[78] B. Glavic *et al.*, "Data provenance," *Foundations and Trends® in Databases*, vol. 9, no. 3-4, pp. 209–441, 2021 (cit. on p. 16).

[79] Google, *The size and quality of a data set*, `https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality`, 2022 (cit. on p. 176).

[80] T. Grosse-Puppendahl, *Leveraging AI technologies for Porsche's future*, `https://www.linkedin.com/pulse/leveraging-ai-technologies-porsches-future-tobias-grosse-puppendahl`, 2018 (cit. on p. 180).

[81] F. Grützmacher, B. Beichler, A. Hein, T. Kirste, and C. Haubelt, "Time and memory efficient online piecewise linear approximation of sensor signals," *Sensors*, vol. 18, no. 6, p. 1672, 2018 (cit. on pp. 15, 22, 43, 48, 70).

[82] V. Gulisano, M. Almgren, and M. Papatriantafilou, "Metis: A two-tier intrusion detection system for advanced metering infrastructures," in *International Conference on Security and Privacy in Communication Systems*, Springer, 2014, pp. 51–68 (cit. on p. 40).

[83] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, and P. Tsigas, "STRETCH: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4221–4238, 2022 (cit. on p. 171).

[84] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 2, 11:1–11:28, 2017, ISSN: 2329-4949 (cit. on pp. 42, 44, 56).

[85] V. Gulisano, D. Palyvos-Giannas, B. Havers, and M. Papatriantafilou, "The role of event-time order in data streaming analysis," in *Proceedings of the ACM International Conference on Distributed and Event-based Systems*, Montreal, Quebec, Canada: ACM, 2020, 214–217, ISBN: 9781450380287 (cit. on p. 12).

[86] V. Gulisano, V. Tudor, M. Almgren, and M. Papatriantafilou, "Bes: Differentially private and distributed event aggregation in advanced metering infrastructures," in *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, ACM, 2016, pp. 59–69 (cit. on p. 40).

[87] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011 (cit. on p. 43).

[88] HardKernel, *Odroid-XU4*, 2020. [Online]. Available: `http://www.hardkernel.com` (cit. on pp. 130, 164).

[89] B. Havers, *Nona - public repository*, 2023. [Online]. Available: `https://github.com/anonymous6e6f6e61/nona/` (cit. on pp. 148, 163).

[90] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A. C. Koppisetty, and M. Papatriantafilou, "DRIVEN: A framework for efficient data retrieval and clustering in vehicular networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, 2019, pp. 1850–1861 (cit. on pp. 50, 70, 78, 92, 95, 107, 108).

[91] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou, and A. C. Koppisetty, "DRIVEN: A framework for efficient data retrieval and clustering in vehicular networks," *Future Generation Computer Systems*, vol. 107, pp. 1–17, 2020 (cit. on pp. 13, 14, 76, 78, 92, 95, 107, 108, 114, 146, 176).

[92] J. He, S. Ji, Y. Pan, and Y. Li, "Constructing load-balanced data aggregation trees in probabilistic wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 7, pp. 1681–1690, 2013 (cit. on p. 107).

[93] I. Hegedűs, G. Danner, and M. Jelasity, "Gossip learning as a decentralized alternative to federated learning," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2019, pp. 74–90 (cit. on p. 177).

[94] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? what form? what from?" *VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017 (cit. on p. 140).

[95] X. Huang, R. Yu, J. Kang, and Y. Zhang, "Distributed reputation management for secure and efficient vehicular edge computing and networks," *IEEE Access*, vol. 5, pp. 25 408–25 420, 2017 (cit. on p. 108).

[96] C. Hubmann, J. Schulz, M. Becker, D. Althoff, and C. Stiller, "Automated driving in uncertain environments: Planning with interaction and uncertain maneuver prediction," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 1, pp. 5–17, 2018 (cit. on p. 176).

[97] B. Hull, V. Bychkovsky, Y. Zhang, *et al.*, "CarTel: A distributed mobile sensor computing system," in *4th Int'l Conf. on Embedded networked sensor systems*, ACM, 2006, pp. 125–138 (cit. on p. 107).

[98] M. Huq, A. Wombacher, and P. Apers, "Adaptive inference of fine-grained data provenance to achieve high accuracy at lower storage costs," in *7th IEEE International Conference on E-Science, e-Science 2011*. USA: IEEE Computer Society, Dec. 2011, pp. 202–209, eemcs-eprint-21400 (cit. on p. 140).

[99] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and reliable stream processing over wide area networks," in *2007 IEEE 23rd International Conference on Data Engineering Workshop*, Apr. 2007, pp. 604–613 (cit. on pp. 13, 118, 119, 149, 150).

[100] IEEE. "IEEE 802.11p-2010 - IEEE Standard for Information technology - Wireless access in vehicular environments." accessed 2020-10-02. (2010), [Online]. Available: `https://standards.ieee.org/standard/802_11p-2010.html` (cit. on p. 8).

[101] S. Ilarri, T. Delot, and R. Trillo-Lado, "A data management perspective on vehicular networks," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2420–2460, 2015 (cit. on p. 95).

[102] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010 (cit. on p. 69).

[103] K. M. Jensen, I. F. Santos, and H. J. Corstens, "Estimation of brake pad wear and remaining useful life from fused sensor system, statistical data processing, and passenger car longitudinal dynamics," *Wear*, p. 205 220, 2023 (cit. on p. 3).

[104] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer, "Quality-driven continuous query execution over out-of-order data streams," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, New York, NY, USA: ACM, 2015, pp. 889–894 (cit. on p. 42).

[105] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6g: A comprehensive survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 334–366, 2021 (cit. on p. 3).

[106] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big Automotive Data: Leveraging large volumes of data for knowledge-driven product development," in *Proceedings of the IEEE International Conference on Big Data*, 2014, pp. 736–741 (cit. on pp. 6, 9).

[107] M. Kakkasageri and S. Manvi, "Information management in vehicular ad hoc networks: A review," *Journal of network and computer Applications*, vol. 39, pp. 334–350, 2014 (cit. on p. 76).

[108] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "THEMIS: Fairness in federated stream processing under overload," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, New York, NY, USA: ACM, 2016, pp. 541–553 (cit. on p. 42).

[109] J. Karimov, T. Rabl, and V. Markl, "AJoin: Ad-hoc stream joins at scale," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 435–448, 2019 (cit. on p. 171).

[110] J. Karimov, T. Rabl, and V. Markl, "AStream: Ad-hoc shared stream processing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 607–622 (cit. on p. 171).

[111] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Proceedings of 2001 IEEE International Conference on Data Mining*, IEEE, 2001, pp. 289–296 (cit. on pp. 42, 48, 70).

[112] A. Keramatian, V. Gulisano, M. Papatriantafilou, P. Tsigas, and Y. Nikolakopoulos, "MAD-C: Multi-stage approximate distributed cluster-combining for obstacle detection and localization," in *European Conference on Parallel Processing*, Springer, 2018, pp. 312–324 (cit. on p. 40).

[113] L Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Proceedings 22nd international conference on distributed computing systems workshops*, IEEE, 2002, pp. 575–578 (cit. on p. 107).

[114] A. Krizhevsky, V. Nair, and G. Hinton, *The CIFAR-10 dataset*, `http://www.cs.toronto.edu/kriz/cifar.html`, 2014 (cit. on p. 185).

[115] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012 (cit. on p. 3).

[116] N. Kumar and J.-H. Lee, "Peer-to-peer cooperative caching for data dissemination in urban vehicular communications," *IEEE Systems Journal*, vol. 8, no. 4, pp. 1136–1144, 2014 (cit. on p. 107).

[117] R. Kumar and M. Dave, "A framework for handling local broadcast storm using probabilistic data aggregation in vanet," *Wireless personal communications*, vol. 72, no. 1, pp. 315–341, 2013 (cit. on p. 107).

[118] Y. Lai, Y. Xu, F. Yang, W. Lu, and Q. Yu, "Privacy-aware query processing in vehicular ad-hoc networks," *Ad Hoc Networks*, vol. 91, p. 101 876, 2019 (cit. on pp. 107, 108).

[119] Y. Lai, F. Yang, J. Su, *et al.*, "Fog-based two-phase event monitoring and data gathering in vehicular sensor networks," *Sensors*, vol. 18, no. 1, p. 82, 2018 (cit. on pp. 76, 108).

[120] Y. Lai, L. Zhang, F. Yang, L. Zheng, T. Wang, and K.-C. Li, "CASQ: Adaptive and cloud-assisted query processing in vehicular sensor networks," *Future Generation Computer Systems*, vol. 94, pp. 237–249, 2019 (cit. on p. 107).

[121] Y. Lai, L. Zheng, T. Wang, F. Yang, and Q. Zhou, "Cloud-assisted data storage and query processing at vehicular ad-hoc sensor networks," in *Int'l Conf. on Security, Privacy and Anonymity in Computation, Communication and Storage*, Springer, 2017, pp. 692–702 (cit. on p. 107).

[122] S. Lee, X. Zheng, J. Hua, H. Vikalo, and C. Julien, "Opportunistic federated learning: An exploration of egocentric collaboration for pervasive computing applications," in *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE, 2021, pp. 1–8 (cit. on p. 176).

[123] U. Lee, J. Lee, J.-S. Park, and M. Gerla, "FleaNet: A virtual market place on vehicular networks," *IEEE Trans. on Vehicular Technology*, vol. 59, no. 1, pp. 344–355, 2010 (cit. on p. 107).

[124] U. Lee, E. Magistretti, M. Gerla, P. Bellavista, and A. Corradi, "Dissemination and harvesting of urban data using vehicular sensing platforms," *IEEE transactions on vehicular technology*, vol. 58, no. 2, pp. 882–901, 2008 (cit. on p. 107).

[125] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: A new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008 (cit. on pp. 119, 150).

[126] J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *Proceedings of the 8th ACM SIGMOD workshop on Research Issues in Data Mining and Knowledge Discovery*, ACM, 2003, pp. 2–11 (cit. on p. 15).

[127] J. Lin, E. Keogh, L. Wei, and S. Lonardi, "Experiencing SAX: A novel symbolic representation of time series," *Data Mining and knowledge discovery*, vol. 15, no. 2, pp. 107–144, 2007 (cit. on p. 14).

[128] J. Lin, M. Vlachos, E. Keogh, *et al.*, "A MPAA-based iterative clustering algorithm augmented by nearest neighbors search for time-series data streams," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2005, pp. 333–342 (cit. on p. 70).

[129] C.-L. Liu, C.-Y. Wang, and H.-Y. Wei, "Cross-layer mobile chord P2P protocol design for VANET," *Int'l Journal of Ad Hoc and Ubiquitous Computing*, vol. 6, no. 3, pp. 150–163, 2010 (cit. on pp. 24, 107).

[130] L. Liu, C. Chen, Q. Pei, S. Maharjan, and Y. Zhang, "Vehicular edge computing and networking: A survey," *Mobile Networks and Applications*, pp. 1–24, 2020 (cit. on pp. 9, 76).

[131] P. A. Lopez, M. Behrisch, L. Bieker-Walz, *et al.*, "Microscopic traffic simulation using SUMO," in *The 21st IEEE International Conference on Intelligent Transportation Systems*, IEEE, 2018 (cit. on pp. 26, 177).

[132] S. Lu, Y. Yao, and W. Shi, "Collaborative learning on the edges: A case study on connected vehicles," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019 (cit. on pp. 76, 176).

[133] Luminar, *Technology*, 2023. [Online]. Available: `https://www.luminartech.com/technology` (cit. on p. 6).

[134] G. Luo, K. Yi, S.-W. Cheng, *et al.*, "Piecewise linear approximation of streaming time series data with max-error guarantees," in *2015 IEEE 31st International Conference on Data Engineering*, IEEE, 2015, pp. 173–184 (cit. on pp. 43, 70).

[135] I. Markit, *These OEMs are launching 5G-enabled cars years before the tech goes mainstream*, 2021. [Online]. Available: `https://ihsmarkit.com/research-analysis/these-oems-are-launching-5genabled-cars-years-before-the-tech-.html` (cit. on p. 179).

[136] Max Peterson et. al. "BADA – on-board off-board distributed data analytics." accessed 2020-10-03. (2020), [Online]. Available: `https://www.vinnova.se/globalassets/mikrosajter/ffi/dokument/slutrapporter-ffi/effektiva-och-uppkopplade-transporter-rapporter/2016-04260eng.pdf` (cit. on p. 9).

[137] J. C. McCallum, *Our World in Data - historical cost of computer memory and storage*, 2022. [Online]. Available: `https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage` (cit. on p. 4).

[138] McKinsey. "Car data: Paving the way to value-creating mobility." accessed 2020-08-29. (2016), [Online]. Available: `https://www.mckinsey.com/~/media/McKinsey/Industries/Automotive\%20and\%20Assembly/Our\%20Insights/Creating\%20value\%20from\%20car\%20data/Creating\%20value\%20from\%20car\%20data.pdf` (cit. on p. 7).

[139] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*, PMLR, 2017, pp. 1273–1282 (cit. on pp. 18, 76, 179, 182, 184).

[140] D. Milojicic, "The Edge-to-Cloud Continuum," *Computer*, vol. 53, no. 11, pp. 16–25, 2020, ISSN: 1558-0814 (cit. on p. 3).

[141] MongoDB, *MongoDB*, 2020. [Online]. Available: `https://www.mongodb.com` (cit. on p. 138).

[142] H. Najdataei, Y. Nikolakopoulos, V. Gulisano, and M. Papatriantafilou, "Continuous and parallel LiDAR point-cloud clustering," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2018, pp. 671–684 (cit. on pp. 27, 41, 43, 46, 50, 52, 53, 114).

[143] Neo4j, *Neo4j*, 2020. [Online]. Available: `https://neo4j.com/` (cit. on p. 138).

[144] A. Nshimiyimana, D. Agrawal, and W. Arif, "Comprehensive survey of V2V communication for 4G mobile and wireless technology," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, 2016, pp. 1722–1726 (cit. on p. 8).

[145] NVIDIA. "NVIDIA Drive AGX Orin." (2023), [Online]. Available: `https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orin-advanced-software-defined-platform-for-autonomous-machines` (cit. on pp. 6, 8).

[146] Ofcom, *Measuring mobile broadband performance in the UK - 4G and 3G network performance*, `http://static.ofcom.org.uk/static/research/mbb.pdf`, 2014 (cit. on p. 95).

[147] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran, "MCEP: A mobility-aware complex event processing system," *ACM Transactions on Internet Technology (TOIT)*, vol. 14, no. 1, p. 6, 2014 (cit. on p. 42).

[148] S. Ozdemir and Y. Xiao, "Secure data aggregation in wireless sensor networks: A comprehensive overview," *Computer Networks*, vol. 53, no. 12, pp. 2022–2037, 2009 (cit. on p. 107).

[149] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016 (cit. on p. 176).

[150] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "GeneaLog: Fine-grained data streaming provenance at the edge," in *Proceedings of the 19th International Middleware Conference*, ACM, 2018, pp. 227–238 (cit. on pp. 16, 40, 76, 151, 171).

[151] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "GeneaLog: Fine-grained data streaming provenance in cyber-physical systems," *Parallel Computing*, vol. 89, p. 102 552, 2019 (cit. on pp. 24, 25, 114, 115, 119, 130–132, 140).

[152] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Haren: A framework for ad-hoc thread scheduling policies for data streaming applications," in *13th ACM Int'l Conf. on Distributed Event-Based Systems (DEBS)*, ACM, Jun. 2019, pp. 19–30 (cit. on pp. 76, 114).

[153] D. Palyvos-Giannas, B. Havers, M. Papatriantafilou, and V. Gulisano, "Ananke: A streaming framework for live forward provenance," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 391–403, 2020 (cit. on pp. 76, 146, 150, 151, 153, 154, 156, 162–164, 171, 176).

[154] D. Palyvos-Giannas, K. Tzompanaki, M. Papatriantafilou, and V. Gulisano, "Erebus: Explaining the outputs of data streaming queries," in *Very Large Data Base*, vol. 16, 2023, pp. 230–242 (cit. on p. 16).

[155] G. Pandey, J. R. McBride, and R. M. Eustice, "Ford campus vision and lidar data set," *The International Journal of Robotics Research*, vol. 30, no. 13, pp. 1543–1552, 2011 (cit. on pp. 12, 46, 53).

[156] F. Pasqualetti, F. Dörfler, and F. Bullo, "Attack detection and identification in cyber-physical systems," *IEEE Transactions on Automatic Control*, vol. 58, no. 11, pp. 2715–2729, 2013 (cit. on p. 114).

[157] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017 (cit. on p. 183).

[158] J. Posner, L. Tseng, M. Aloqaily, and Y. Jararweh, "Federated learning in vehicular networks: Opportunities and solutions," *IEEE Network*, vol. 35, no. 2, pp. 152–159, 2021 (cit. on p. 176).

[159] PostgreSQL, *PostgreSQL*, 2020. [Online]. Available: `https://www.postgresql.org` (cit. on p. 137).

[160] R. Prytz, S. Nowaczyk, T. Rögnvaldsson, and S. Byttner, "Predicting the need for vehicle compressor repairs using maintenance records and logged vehicle data," *Engineering applications of artificial intelligence*, vol. 41, pp. 139–150, 2015 (cit. on p. 7).

[161] I. Psaras, O. Ascigil, S. Rene, G. Pavlou, A. Afanasyev, and L. Zhang, "Mobile data repositories at the edge," in *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018 (cit. on p. 13).

[162] PWC. "The 2017 Strategy& Digital Auto Report." accessed 2020-08-29. (2017), [Online]. Available: `https://www.strategyand.pwc.com/gx/en/insights/2017/fast-and-furious/2017-strategyand-digital-auto-report.pdf` (cit. on pp. 7, 8).

[163] Qualcomm, *Cellular-V2X technology overview*, 2019. [Online]. Available: `https://www.qualcomm.com/media/documents/files/c-v2x-technology-overview.pdf` (cit. on p. 180).

[164] R. Rajagopalan and P. K. Varshney, "Data-aggregation techniques in sensor networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 8, no. 4, pp. 48–63, 2006 (cit. on p. 107).

[165] C. A. Ralanamahatana, J. Lin, D. Gunopulos, E. Keogh, M. Vlachos, and G. Das, "Mining time series data," in *Data mining and knowledge discovery handbook*, Springer, 2005, pp. 1069–1103 (cit. on p. 70).

[166] A. Rasouli, I. Kotseruba, and J. K. Tsotsos, "Understanding pedestrian behavior in complex traffic scenes," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 1, pp. 61–70, 2018 (cit. on p. 176).

[167] D. Reinsel, J. Gantz, and Rydning. "The digitization of the world - from edge to core." accessed 2020-09-01. (2018), [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (cit. on p. 3).

[168] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *Science*, vol. 344, no. 6191, pp. 1492–1496, 2014 (cit. on p. 69).

[169] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Comput. Surv.*, vol. 52, no. 2, 2019 (cit. on p. 171).

[170] J. van Rooij, V. Gulisano, and M. Papatriantafilou, "LoCoVolt: Distributed detection of broken meters in smart grids through stream processing," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ACM, 2018, pp. 171–182 (cit. on pp. 40, 114).

[171] K. Roscher, S. Bittl, A. Gonzalez, M Myrtus, and J. Jiru, "ezCar2X: Rapid-prototyping of communication technologies and cooperative ITS applications on real targets and inside simulation environments," in *11th Conference Wireless Communication and Information*, 2014, pp. 51–62 (cit. on pp. 26, 177).

[172] R. B. Rusu, "Semantic 3D object maps for everyday manipulation in human living environments," *KI-Künstliche Intelligenz*, vol. 24, no. 4, pp. 345–348, 2010 (cit. on p. 43).

[173] S. Salah, G. Maciá-Fernández, and J. E. Díaz-Verdejo, "A model-based survey of alert correlation techniques," *Computer Networks*, vol. 57, no. 5, pp. 1289 –1317, 2013 (cit. on p. 114).

[174] S. Savazzi, M. Nicoli, and V. Rampa, "Federated learning with cooperating devices: A consensus approach for massive iot networks," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4641–4654, 2020 (cit. on p. 176).

[175] E. Schoch, F. Kargl, M. Weber, and T. Leinmuller, "Communication patterns in VANETs," *IEEE Communications*, vol. 46, no. 11, pp. 119–125, 2008 (cit. on pp. 7, 76).

[176] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, e4257, 2017 (cit. on p. 165).

[177] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. De Carvalho, and J. Gama, "Data stream clustering: A survey," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 13, 2013 (cit. on p. 69).

[178] C. Sommer, D. Eckhoff, A. Brummer, *et al.*, "Veins: The open source vehicular network simulation framework," in *Recent advances in network simulation*, Springer, 2019, pp. 215–252 (cit. on pp. 26, 177, 182).

[179] SQLite, *SQLite*, 2020. [Online]. Available: https://www.sqlite.org/ (cit. on p. 137).

[180] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005 (cit. on pp. 12, 41, 42, 114, 146).

[181] T. Suel, "Delta compression techniques," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds. Springer International Publishing, 2018, pp. 1–8, ISBN: 978-3-319-63962-8 (cit. on p. 14).

[182] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of the 29th international Conference on Very Large Data Bases-Volume 29*, VLDB Endowment, 2003, pp. 309–320 (cit. on p. 42).

[183] P. Taylor, *Statista - cellular network average speed in US 2016-2023*, 2023. [Online]. Available: `https://www.statista.com/statistics/995096/average-cellular-network-speed-in-the-us/` (cit. on p. 3).

[184] P. Taylor, *Statista - data growth worldwide*, 2023. [Online]. Available: `https://www.statista.com/statistics/871513/worldwide-data-created/` (cit. on pp. 3, 4).

[185] TensorFlow, *TensorFlow Federated*, `https://www.tensorflow.org/federated`, 2022 (cit. on p. 177).

[186] S.-L. Tsao and C.-M. Cheng, "Design and evaluation of a two-tier peer-to-peer traffic information system," *IEEE Communications*, vol. 49, no. 5, pp. 165–172, 2011 (cit. on p. 107).

[187] L. Ulanova, N. Begum, M. Shokoohi-Yekta, and E. Keogh, "Clustering in the face of fast changing streams," in *Proceedings of 2016 SIAM International Conference on Data Mining*, SIAM, 2016, pp. 1–9 (cit. on p. 69).

[188] G. Ulm, E. Gustavsson, and M. Jirstrand, "OODIDA: On-board/off-board distributed data analytics for connected vehicles," *arXiv preprint arXiv:1902.00319*, 2019 (cit. on p. 76).

[189] A. Varga, "OMNeT++," in *Modeling and tools for network simulation*, Springer, 2010, pp. 35–59 (cit. on pp. 26, 177).

[190] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE '10, New York, NY, USA: Association for Computing Machinery, 2010 (cit. on p. 138).

[191] N. N. Vijayakumar and B. Plale, "Towards low overhead provenance tracking in near real-time stream filtering," in *Provenance and Annotation of Data: International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, Springer, 2006, pp. 46–54 (cit. on pp. 140, 171).

[192] S. Wagner and D. Wagner, *Comparing clusterings: an overview*. Universität Karlsruhe, Fakultät für Informatik Karlsruhe, 2007 (cit. on p. 54).

[193] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE, 2006, pp. 61–69 (cit. on p. 50).

[194] H. Wang, Z. Kaplan, D. Niu, and B. Li, "Optimizing federated learning on non-iid data with reinforcement learning," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 1698–1707 (cit. on pp. 177, 183).

[195] M. Wang, M. Blount, J. Davis, A. Misra, and D. Sow, "A time-and-value centric provenance model and architecture for medical event streams," in *Proceedings of the 1st ACM SIGMOBILE International Workshop on Systems and Networking Support for Healthcare and Assisted Living Environments*, ser. HealthNet '07, New York, NY, USA: ACM, 2007, pp. 95–100 (cit. on p. 140).

[196] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *Ieee Access*, vol. 5, pp. 6757–6779, 2017 (cit. on p. 76).

[197] T. Wang, L. Song, and Z. Han, "Collaborative data dissemination in cognitive vanets with sensing-throughput tradeoff," in *1st IEEE Int'l Conf. on Communications in China (ICCC)*, IEEE, 2012, pp. 41–45 (cit. on pp. 24, 107).

[198] J. S. Weber, M. Neves, and T. Ferreto, "VANET simulators: An updated review," *Journal of the Brazilian Computer Society*, vol. 27, no. 1, pp. 1–31, 2021 (cit. on p. 177).

[199] B. Wire, *Strategy analytics: Mobile data revenue falls below US$1 per gigabyte as 5G uplift proves elusive*, `https://www.businesswire.com/news/home/20210413005861/en/Strategy-Anal`, 2021 (cit. on p. 176).

[200] World Economic Forum. "Big data, big impact: New possibilities for international development." accessed 2020-09-01. (2012), [Online]. Available: `http://www3.weforum.org/docs/WEF_TC_MFS_BigDataBigImpact_Briefing_2012.pdf` (cit. on p. 3).

[201] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng, "Maximum error-bounded piecewise linear representation for online stream approximation," *The VLDB Journal*, vol. 23, no. 6, pp. 915–937, 2014 (cit. on pp. 43, 48, 70).

[202] J. Xu, Z. Jin, M. Xu, and N. Zheng, "Mobile-aware anonymous peer selecting algorithm for enhancing privacy and connectivity in location-based service," in *7th Int'l Conf. on E-Business Engineering*, IEEE, 2010, pp. 172–177 (cit. on p. 108).

[203] W. Xu, H. Zhou, N. Cheng, *et al.*, "Internet of vehicles in Big Data era," *IEEE/CAA Journal of Automatica Sinica*, vol. 5, no. 1, pp. 19–35, 2018 (cit. on p. 6).

[204] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM Sigmod record*, vol. 31, no. 3, pp. 9–18, 2002 (cit. on p. 107).

[205] S. Yousefi, M. S. Mousavi, and M. Fathy, "Vehicular ad hoc networks (VANETs): Challenges and perspectives," in *Proceedings of 2006 6th International Conference on ITS Telecommunications*, IEEE, 2006, pp. 761–766 (cit. on p. 40).

[206] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58 443–58 469, 2020 (cit. on p. 3).

[207] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Maximizing determinism in stream processing under latency constraints," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ACM, 2017, pp. 112–123 (cit. on p. 42).

[208] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments.," in *Osdi*, vol. 8, 2008, p. 7 (cit. on p. 108).

[209] K. Zhang, Y. Mao, S. Leng, A. Vinel, and Y. Zhang, "Delay constrained offloading for Mobile Edge Computing in cloud-enabled vehicular networks," in *Proceedings of the International Workshop on Resilient Networks Design and Modeling*, IEEE, 2016, pp. 288–294 (cit. on p. 9).

[210] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden, "ICEDB: Intermittently-connected continuous query processing," in *23rd IEEE Int'l Conf. on Data Engineering (ICDE 2007)*, IEEE, 2007, pp. 166–175 (cit. on p. 107).

[211] Y. Zhang, J. Zhao, and G. Cao, "Roadcast: A popularity aware content sharing scheme in vanets," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 13, no. 4, pp. 1–14, 2010 (cit. on p. 107).

[212] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018 (cit. on p. 76).

[213] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma, "Understanding mobility based on GPS data," in *Proceedings of the 10th International Conference on Ubiquitous Computing*, ACM, 2008, pp. 312–321 (cit. on p. 53).

[214] Y. Zheng, X. Xie, and W.-Y. Ma, "Geolife: A collaborative social networking service among user, location and trajectory.," *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 32–39, 2010 (cit. on pp. 53, 92, 134).

[215] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from GPS trajectories," in *18th Int'l Conf. on World wide web*, ACM, 2009, pp. 791–800 (cit. on p. 53).

[216] J. Zhou, R. Q. Hu, and Y. Qian, "Scalable distributed communication architectures to support advanced metering infrastructure in smart grid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 9, pp. 1632–1642, 2012 (cit. on p. 40).