Thesis for The Degree of Doctor of Philosophy

Efficient Data Streaming Analytic Designs for Parallel and Distributed Processing

Hannaneh Najdataei

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Efficient Data Streaming Analytic Designs for Parallel and Distributed Processing

Hannaneh Najdataei

© Hannaneh Najdataei, 2022
except where otherwise stated.
All rights reserved.

ISBN 978-91-7905-706-0
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 5172.
ISSN 0346-718X
Technical Report No. 226D

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

This thesis has been prepared using LaTeX.
The figures in this thesis have been designed using resources from Flaticon.com.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

In the loving memory of my father.

To my mother with eternal appreciation.

Efficient Data Streaming Analytic Designs for Parallel and Distributed Processing

Hannaneh Najdataei

Department of Computer Science and Engineering, Chalmers University of Technology

Abstract

Today, ubiquitously sensing technologies enable inter-connection of physical objects, as part of Internet of Things (IoT), and provide massive amounts of data streams. In such scenarios, the demand for timely analysis has resulted in a shift of data processing paradigms towards continuous, parallel, and multi-tier computing. However, these paradigms are followed by several challenges especially regarding analysis speed, precision, costs, and deterministic execution. This thesis studies a number of such challenges to enable efficient continuous processing of streams of data in a decentralized and timely manner.

In the first part of the thesis, we investigate techniques aiming at speeding up the processing without a loss in precision. The focus is on continuous machine learning/data mining types of problems, appearing commonly in IoT applications, and in particular continuous clustering and monitoring, for which we present novel algorithms; (i) *Lisco*, a sequential algorithm to cluster data points collected by LiDAR (a distance sensor that creates a 3D mapping of the environment), (ii) *p-Lisco*, the parallel version of *Lisco* to enhance pipeline- and data-parallelism of the latter, (iii) *pi-Lisco*, the parallel and incremental version to reuse the information and prevent redundant computations, (iv) *g-Lisco*, a generalized version of *Lisco* to cluster any data with spatio-temporal locality by leveraging the implicit ordering of the data, and (v) *Amble*, a continuous monitoring solution in an industrial process.

In the second part, we investigate techniques to reduce the analysis costs in addition to speeding up the processing while also supporting deterministic execution. The focus is on problems associated with availability and utilization of computing resources, namely reducing the volumes of data, involving concurrent computing elements, and adjusting the level of concurrency. For that, we propose three frameworks; (i) *DRIVEN*, a framework to continuously compress the data and enable efficient transmission of the compact data in the processing pipeline, (ii) *STRATUM*, a framework to continuously pre-process the data before transferring the later to upper tiers for further processing, and (iii) *STRETCH*, a framework to enable instantaneous elastic reconfigurations to adjust intra-node resources at runtime while ensuring determinism.

The algorithms and frameworks presented in this thesis contribute to an efficient processing of data streams in an online manner while utilizing available resources. Using extensive evaluations, we show the efficiency and achievements of the proposed techniques for IoT representative applications that involve a wide spectrum of platforms, and illustrate that the performance of our work exceeds that of state-of-the-art techniques.

Keywords: continuous analysis, stream processing, scalability, elasticity

Acknowledgment

I would like to express my sincere gratitude to my supervisors Marina Papatriantafilou, Vincenzo Gulisano and Philippas Tsigas. The challenging journey of PhD became precious and enjoyable with uplifting discussions, continuous support and generous guidance of these great mentors.

I am also grateful to my co-authors and collaborators that contributed to the work in this thesis. Special thanks to Yiannis, Ivan, Alessandro, Bastian, and Dimitris. I'd like to thank my past and current colleagues at the department for providing a friendly environments. Thank you Aljoscha, Amir, Aras, Babis, Christos, Elad, Fazeleh, Georgia, Iosif, Karl, Magnus, Nasser, Olaf, Romaric, Thomas R., Thomas P., Tomas, Valentin. A special thanks to the department's administration for their support: Eva, Rebecca, Marianne, Rolf, and Clara.

I am honoured to have Prof. Valeria Cardellini as the faculty opponent of the defence and professors Christoph Kessler, Luís Antunes Veiga, Roman Vitenberg, and Pedro Petersen Moura Trancoso on the grading committee. Many thanks to my examiner Jan Jonsson and the follow-up groups that supported me during my studies.

I wish to acknowledge the financial support by the Swedish Foundation for Strategic Research, project Factories in the Cloud (FiC), with grant number GMT14-0032.

I would like to thank my family to whom I owe a great deal. To my great inspiration, my late father, who thought me how to always find a way to think positive, encouraged me to start this PhD, and told me he is proud of me when I needed the most. To my role model in life and work, my mother, who educated me and thought me science, showed me true love, and made countless scarifies for me to follow the path I wanted. To my lovely sister and my wonderful brother to be always there for me. To my friends in Gothenburg and the rest of the world, for providing happy distractions.

Finally, a great appreciation and enormous thanks goes to my husband and the love of my life, Alireza, for his sympathetic ear, unlimited patience, and believing in me. Without his support, I am sure this thesis would have never been completed. The last word goes for Atrin, my baby girl, who has been the light of my life for the last year and who has made me stronger, better and more fulfilled than I could have ever imagined. I love you darling.

Hannaneh Najdataei
Göteborg, August 2022

List of Publications

This thesis is based on the results that are documented in the following publications:

Paper A **H. Najdataei**, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, *"Continuous and parallel lidar point-cloud clustering," In the Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS), pp. 671-684, 2018.*

Paper B **H. Najdataei**, V. Gulisano, P. Tsigas, M. Papatriantafilou, *"pi-Lisco: parallel and incremental stream-based point-cloud clustering," In the Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC), pp. 460-469, 2022.*

Paper C B. Havers, R. Duvignau, **H. Najdataei**, V. Gulisano, M. Papatriantafilou, A. C. Koppisetty, *"DRIVEN: A framework for efficient Data Retrieval and clustering in Vehicular Networks," In the Journal of Future Generation Computer Systems 107, pp. 1-17, 2020*
The above extends the work and results that previously appeared in:
B. Havers, R. Duvignau, **H. Najdataei**, V. Gulisano, A. C. Koppisetty, M. Papatriantafilou, *"Driven: a framework for efficient data retrieval and clustering in vehicular networks," In the Proceedings of the 35th International Conference on Data Engineering (ICDE), pp. 1850-1861, 2019.*

Paper D **H. Najdataei**, M. Subramaniyan, V. Gulisano, A. Skoogh, M. Papatriantafilou, *"Adaptive Stream-based Shifting Bottleneck Detection in IoT-based Computing Architectures," In the Proceedings of the 24th International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 993-1000, 2019.*

Paper E V. Gulisano, **H. Najdataei**, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, P. Tsigas, *"STRETCH: Virtual Shared-Nothing Parallelism for Scalable and Elastic Stream Processing," In the Journal of Transactions on Parallel and Distributed Systems, 2022*
The above extends the work and results that previously appeared in:
H. Najdataei, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, V. Gulisano, *"Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism," In the Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS), pp. 7-18, 2019.*

Other Papers

I have also contributed to the following articles during my PhD studies. However, they are not appended to this thesis, due to contents overlapping with appended publications or not directly in the focus of the thesis.

- **H. Najdataei**, *"Swarm-Intelligence-based Multi-Robot Task Allocation," Poster presented at the 4th ACM Celebration of Women in Computing (womENcourage), 2017.*
- **H. Najdataei**, M. Subramaniyan, V. Gulisano, A. Skoogh, M. Papatriantafilou, *"Stream-IT: Continuous and dynamic processing of production systems data-throughput bottlenecks as a case-study," In the Proceedings of the 28th International Symposium on Industrial Electronics (ISIE), pp. 1328-1333, 2019.*
- O. Bodunov, V. Gulisano, **H. Najdataei**, Z. Jerzak, A. Martin, P. Smirnov, M. Strohbach, H. Ziekow, *"The DEBS 2019 grand challenge," In the Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems (DEBS), pp. 205-208. 2019.*
- V. Gulisano, D. Jorde, R. Mayer, **H. Najdataei**, D. Palyvos-Giannas, *"The DEBS 2020 grand challenge," In the Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems (DEBS), pp. 183-186. 2020.*

Personal Contribution

I am the main author of **Paper A**, **Paper B**, and **Paper D**. I designed, implemented, and evaluated the algorithms proposed therein and also led the writing of the papers themselves. In **Paper C**, I contributed to the design of *g-Lisco* in collaboration with all other authors of the paper. **Paper E** builds on previous work conducted within my research group on efficient stream processing through shared-memory. My novel contribution is the design and implementation of a general-purpose stateful operator rather than a specific one (as in previous work). I also designed and performed extensive benchmarks to study the performance of *STRETCH*, in addition to co-authoring.

Contents

Part I

Thesis Overview

Introduction

"The world's most valuable resource is no longer oil, but data" – The economist[1]

From the first industrial revolution that introduced water and steam power to reduce hard manual labor, to the invention of mass production and assembly lines using electricity in the second, the third industrial revolution shifted the emphasis from analog and mechanical technology to digital and automation with the adoption of computers. Today, we are in the midst of the fourth industrial revolution, a.k.a. *Industry 4.0*, that optimizes the computerization from recent decades with the use of Internet technologies to create smart production and smart services [1].

Industry 4.0 takes what was started in the third revolution and enhances it with smart and autonomous systems through the introduction of *Cyber Physical Systems* (CPS). The term CPS refers to automated systems with integrated computational and physical capabilities. Unlike traditional embedded systems that are designed to be used as stand alone devices, CPS focus on networking several devices and enabling machine-to-machine communication [2]. This vision has led to the emergence of *Internet-of-Things* (IoT) paradigm [3]. IoT allows everyday objects to communicate with one another over the Internet to achieve some useful objective without human involvement. The result of these trends is a new set of applications with a range of requirements, from low latency control loops that quickly react to incoming data, to high throughput processing of aggregated data from connected devices. By 2025, the number of *IoT devices* (e.g. smart watches, smart phones, etc.) installed world wide is estimated to grow to more than 75 billion [4]. The enormous collection of connected devices in IoT makes a significant contribution to the volume of data collected. This massive data, nonetheless, is not lucrative by its own but the analytics offer value [5].

Over the past decades, *machine learning* (ML) and data mining techniques have been widely adopted in a number of complex data-intensive fields to mine the information hidden in the data [6]. However, the characteristics of the continuously collected data (often referred to as *Big Data*) such as high volume and high rate, make the traditional ML algorithms inefficient and impractical. This is because, for instance, most traditional ML algorithms are designed for data to be completely loaded into memory which does not hold in the context of Big Data [7]. Furthermore, *cloud computing* has been a support for Big Data scenarios and the most popular solution to on-demand processing of IoT data [8]. However, the dramatic escalation in data volumes and data rates pushes the limits of centralized data processing infrastructures

[1] https:www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data

for deriving timely intelligence [9]. In the case of IoT applications, the amount of data generated by each device, as well as the need to immediately extract knowledgeable information from such data, make it inefficient and impractical to send all the traffic to the cloud, doing the processing there, and then receive back the results.

The aforementioned inefficiency of traditional data analytics, has begun a new wave of scientific revolution and led to innovative tools such as *continuous processing* that analyses flows of data on the fly. This thesis investigates how algorithms and analytical frameworks, with a focus on CPS, can be adapted to perform continuous data processing in an efficient and decentralized manner. In the next section, we overview the opportunities and challenges introduced by continuous processing of IoT Big Data.

1 Motivation

The emergence of CPS and Industry 4.0 have rapidly accelerated the growth of data and made Big Data a trend in industry [2]. This gives immense opportunities to the research community to introduce innovative, practical, and efficient analytics to help the industry achieve its goals. Big data analytics is a form of advanced data mining (i.e. Knowledge Discovery from Data [10]) that has progressed gradually to turn a large collection of data into knowledge. To distinguish Big Data analytics from conventional ones, it is first necessary to define Big Data and its features.

1.1 Big Data Features

By hearing the term Big Data, normally the first impression is about its size. However, it involves several dimensions while size is only one. Although there has been a divergent discourse on the exact definitions for Big Data during past years, all share a few dimensions based on the "three Vs" suggested by Doug Laney [11]. Laney suggested the Volume, Velocity, and Variety, as three dimensions of challenges in data management. Later, Gartner [12] gave a more detailed definition as:

> "Big Data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation."

In the following, each dimension is described briefly.

Volume refers to the size of the data. As also appeared in the name, Big Data is enormous amount of data. It is, nevertheless, impractical to define a threshold for the size of a dataset to be considered as Big Data. The reason is that the volume dimension along the other dimensions, i.e. velocity and variety, categorizes a dataset as Big Data. In some cases (e.g. sensor readings), data is in form of unbounded streams with infinite size which can be also categorized as Big Data.

Velocity refers to the speed of generating data. In recent years, using the well developed and ubiquitous technological tools, the generating rate of data is unprecedented. However, the term of velocity refers to not only the speed of incoming data, but also the speed that the data should be processed. Therefore, due to the increasing rate at which data is generated, there is a growing demand for processing data immediately as it streams from its sources [13].

Variety refers to the range of the data types. Big Data covers the various data types of the spectrum from fully structured (e.g. tabular data) that can be easily sorted and stored, to unstructured (e.g. video, audio) that is difficult for the machine to analyze. The high variety of the data is what makes data "really big" [14].

1.2 Computing Continuum

Big Data has arguably changed the direction of the development of the hardware architecture as well as software [15]. Motivated by scalability demands of analytical methods, the number of cores in processors is being increased to support scalable parallel computing. Moreover, cloud computing was introduced as one step towards revolutionizing distributed computing to enable convenient and on-demand access to a shared pool of resources (e.g. servers, storage). The concept of cloud computing is about providing nearly unlimited computing capacity and storage by sharing and merging the resources, on request. In IoT applications, the main motivation behind employing clouds, consisting of high-end servers, is to carry out heavy data analysis. This is because IoT devices are often equipped with reduced computational power, i.e. they are resource-constrained devices, and hence likely to be less efficient in performing heavy analysis.

The benefits of cloud computing are nonetheless followed by challenges. The first challenge is that, when dealing with Big Data, it might be impossible to send all data to the cloud without exhausting the available communication bandwidth. Also, sending data to the cloud for analysis and getting back the results to make further decisions, could cause significant high latency which is not tolerable for certain applications requiring real-time processing. Furthermore, such high latency causes a drastic effect on power and energy consumption and influences the reliability of the system [16]. This in turn, would be an indication of degradation in the *Quality-of-Service* (QoS).

The difficulties of shifting all data to the cloud for analysis led to the emergence of *fog computing* and *edge computing* in which the processing procedures are being pushed down closer to where data originates. Edge computing puts the computing at the proximity of data sources. In this paradigm, the intelligence and data processing is offloaded onto the edge devices (e.g. routers, switches, sensors and actuators.) without being sent to the cloud [17]. Edge computing is especially beneficial for applications which require ultra-low latency and real-time analysis. Moreover, edge computing can utilize the communication bandwidth by pre-processing data and making it much more compact than raw information before transferring it to the upper layers.

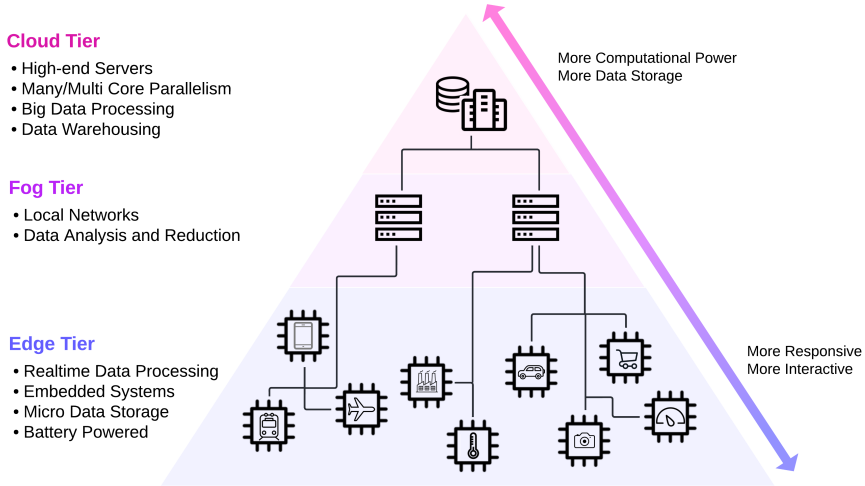

Figure 1: An example of the 3-tier architecture for IoT applications. It includes a cloud tier with high-end servers, a fog tier closer to data origin, and an edge tier with resource constrained devices.

Similar to edge computing, fog computing is pushing the analysis from cloud servers down, closer to the source of data. In fog computing, the analysis is performed on fog nodes (e.g. gateways and small computing servers) that reside near the edge devices and not necessarily on the device itself. By bringing the intelligence away from the cloud, each fog node performs data processing closer to the IoT devices and consequently reduces the amount of data transported. A fog node also can be seen as a mini-cloud, which is located near the edge layer and the IoT devices connected to [18].

Bringing cloud, fog, and edge computing together results in a 3-tier architecture, as shown in Figure 1, where the distribution of data processing is enabled along the things-to-cloud *computing continuum*. The edge/fog/cloud architecture brings several benefits as it allows applications to distribute the intelligence and take advantage of a diverse range of computing resources and storing assets. Nevertheless, in order to design efficient algorithms and frameworks to process data in such 3-tier architecture we need to know *how* and *where* to process the data.

How to process the data? The increasing speed of the data generation demands some form of analytics to process data before it becomes too big. ML is a fundamental component of such data analytics which gives computers the ability to learn from data, provide data driven insights, and make decisions. Nevertheless, the high rate and unbounded nature of data streams challenge traditional ML algorithms to uncover fine-grained patterns and perform timely analysis [19]. To this end, regardless of which layer of the 3-tier architecture the

analysis is running on, there is a need to adapt the traditional ML algorithms to continuously process the flow of data.

Continuous processing allows improvements in memory access patterns as well as enabling real-time monitoring which is a requirement for many applications (e.g. e-commerce order processing, failure detection, air traffic control). The ability of continuous processing in performing analysis on data in motion, as the latter is received, makes it possible to generate results continuously and quickly, thus, significantly benefits applications with small delay constraints. For example, by using continuous processing and querying the constant stream of data received from a temperature sensor, it is possible to raise an alarm once the temperature reaches a certain threshold.

Where to process the data? As data processing requirements grow, IoT applications demand utilization of the whole spectrum of devices in the computing continuum, from cloud data centers to edge systems and endpoint devices. This, in turn, requires hardware-aware processing approaches that can be efficiently deployed at multiple tiers, taking into account the application characteristics as well as the hardware capabilities and constraints. To this end, many research works (e.g. [20] and [21]) conduct performance analysis of the resources across the layers to support the decision process of where to compute applications. Based on such recommendations, for instance, a machine learning application with a neural network consisting of multiple layers and large data sets is a better option for cloud computing while a collaborative data filtering can be run on fog nodes. Moreover, edge computing is recommended for applications with soft execution time constraints to reduce the energy costs and carbon footprints [20].

The heterogeneity of the computing continuum provides opportunities for applications with different requirements where a fluid integration of the resources can be leveraged to support data-driven workflow [22]. Figure 2 illustrates an example of smart transportation system that utilizes edge, fog, and cloud computing. As shown in the figure, the data regarding street conditions and unexpected events such as accidents are generated by many sensors deployed in the cars. This data, then, can be processed locally by resource constrained devices next to the sensors, to be used by the cars to adjust the velocity and avoid obstacles (edge computing). Cars can also send such data to the nearby traffic lights to be processed by the small embedded servers and extract information in order to adjust the lights, thus, relieve potential traffic congestion around the reported location (fog computing). Moreover, the city services can receive data from cars as well as the transportation infrastructures including traffic lights, to process the aggregated data for making further decisions (cloud computing).

2 Background

In this section, we overview the background topics of the focus areas of this thesis to facilitate the reading.

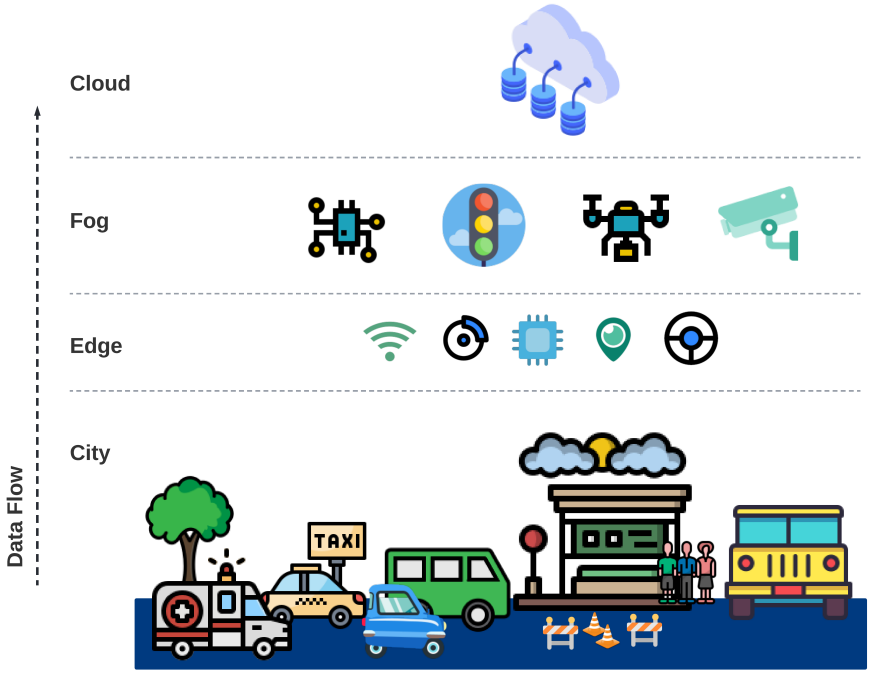

Figure 2: Smart transportation use case with computing options in the edge, fog, and cloud tiers.

2.1 Data Mining Algorithms

The Big Data produced by CPS could transform how we live, work, and think by empowering insight discovery and improving decision making. The realization of this grand potential relies on the ability to extract value through data analytics and identify hidden patterns as the first step towards enabling diagnosis tools [23] (e.g. for detection-based predictive maintenance applications). During the last decade, data mining and ML has gained significant attention in applications related to CPS due to the generation of a large amount of data [24]. In the following, we briefly review two problems related to CPS that are also studied in this thesis.

2.1.1 Spatio-Temporal Data Clustering

In Industry 4.0, sensors play an important role to make CPS smarter by observing physical environments and detecting events of interest. Since sensors reflect the nature of the physical systems, there is usually a spatio-temporal relation implicitly defined in the sensed data [25]. For example, a GPS provides spatial and temporal information of a moving object, which makes it possible to classify the transportation mode of car, bus, or walk as well as to predict the most probable route [26]. In these circumstances, it is important to

identify critical events by correlating similar data points, which can be done by *clustering*. Clustering is a core problem in data mining. It is the process of grouping data into sets (i.e. clusters) using similarity metrics, in a way that the intra-cluster similarity is maximized. In general, there are a few typical requirements for clustering [10]. For instance, clusters should be in any size or shape. To motivate this requirement, consider clustering of a sensor's data for a surveillance application, for which the clustering algorithms should be able to detect objects with arbitrary shapes. Moreover, it is preferable for clustering algorithms to not require domain knowledge to determine the parameters such as the desired number of clusters. This requirement is particularly important for applications in which the user still needs to grasp a deep understanding of the data. It is also important that the clustering algorithm is capable of handling noisy data as most real data sets contain outliers and missing data. Furthermore, desirable clustering algorithms are required to update the existing clusters incrementally with respect to the incoming data rather than recomputing new clusters from scratch.

An extensive literature exists about clustering algorithms that differ on what should be considered as a cluster (cf. [27] and references therein). For instance, K-means [28] generates well-balanced ball-like clusters, whereas density based algorithms such as DBSCAN [29] and OPTICS [30], partition a given set based on the density regions. Among widely used approaches, distance based methods are employed in different applications due to the ability of the former to find arbitrarily shaped clusters without requiring to know the number of clusters a priori.

Distance based clustering methods partition spatially isolated regions and potentially detect clusters by using a distance metric. For example, the *Euclidean distance based clustering* algorithm [31] builds the clusters based on the Euclidean distances between data points, as the clustering metric, using two user defined parameters `minPts` and ϵ. The algorithm creates clusters each containing at least `minPts` number of points. Within each cluster, each point is expected to have a distance less than ϵ with at least another point in the cluster. Points that do not belong to any cluster at the end of the procedure are characterized as noise.

2.1.2 Bottleneck Detection

One of the main indicators of system performance in industrial processes that is often required from the analysts to monitor, is the throughput of the production system [32]. Throughput is commonly affected by arbitrary disruptions in the machines in the system, such as fluctuations in the cycle time, down times and minor stops. These machines are thus referred to as *throughput bottlenecks*. To detect bottlenecks, several methods exist in the literature that use different information (cf. [33] and references therein). In the *average-waiting-time* [34], the machine in which a job waits longest, as measured by the average time a job spends in the queue, is considered the bottleneck. The *active-period* method [35] measures the duration of the periods in which a machine is active without interruption, and calculates the average active period for each machine.

The machine which has the longest average active period is considered the bottleneck. The *shifting-bottleneck detection* method [36] uses the active period of machines to detect momentary bottlenecks of a production system and identifies sole and shifting bottlenecks over a selected interval of time (e.g. a production run hour, shift, day, week). Detecting shifting bottlenecks is particularly important when there is a correlation between machines in a production system. In such systems, there is a higher probability of bottlenecks shifting from one machine to the other during different production runs due to various reasons such as random maintenance stops in the machine, random processing times, and so on.

2.2 Aspects of Big Data Processing

As mentioned before, Big Data has several dimensions which make data processing challenging. The challenges arise especially when the volume or velocity of data overwhelms the processing system, which in turn results in degradation of performance. In this regard, to address the challenges and improve the performance of the Big Data processing, efficient analytics need to adapt various scales [37]:

- ***scale down*** *the amount of data processed.*
  The first step to effectively process Big Data is to reduce its massive size. Data reduction methods include algorithms for pre-processing, redundancy elimination, compression based data reduction, dimension reduction, etc [38]. Another useful technique for data reduction is approximation. This technique is especially important for applications that require analytics to be fast even if it comes with the price of a lower precision, rather than being exact with long waiting time.

- ***scale up*** *the computing resources on a node.*
  Scaling up and employing additional cores on a node to analyze the data is the next step towards tackling the challenges in Big Data processing. Today, thanks to decades of research that provided current modern computing platforms, scaling up (i.e. *vertical scaling*) can improve processing performance significantly due to availability of high computational power and memory in each node. Exploiting available resources is especially beneficial for powerful computing nodes deployed in the cloud layer that enable shared memory and parallel computing.

- ***scale out*** *the computing to distributed nodes.*
  When a single node with all its available resources is not capable of performing efficient processing, the computing should be scaled out to other nodes. Scaling out (i.e. *horizontal scaling*) involves workload distribution over several nodes in the cloud, fog, or edge layer. In this kind of scaling, multiple independent machines are employed to enhance data processing power.

2.2.1 On Data Compression

Due to the increasing amount of data being collected at the edge, scaling down techniques such as data compression are being leveraged and further developed to significantly reduce the amount of data. In this sense, some compression algorithms are designed to support the exact reconstruction of the original data after decompression (i.e. lossless compression) while other algorithms reconstruct data with an approximation of the original information (i.e. lossy compression) [39]. Although lossy algorithms may lead to loss of information, they generally ensure some additional gains in terms of time, communication, and energy saving [40]. One of the most commonly used lossy algorithms on the edge devices to continuously reduce the size of the data – which is also leveraged in this thesis – is *Piecewise Linear Approximation* (PLA).

PLA refers to the approximation of a time series T, of length n, with K straight lines (i.e. *segments*). Because K is typically much smaller than n, PLA makes the storage, transmission and computation of the data more efficient [41]. Additionally, many of the techniques employing PLA, consider an approximation error (or maximum 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 while achieving smaller representations compared to lossless algorithms. Recent works on PLA [42] increasingly place the focus on the streaming aspect of the compression process, and advocate low time and memory consumption as well as small latency while achieving a high compression, in order for PLA to be feasibly implemented close to a sensor's stream.

2.2.2 Parallel Processing

The demand for more computing power, encouraged processor designers to improve the performance of a processor by increasing the clock rate. This strategy worked fine until it hit the physical limit (i.e. power wall) which showed a break down in Dennard scaling by indicating that the power dissipation does not scale with the size of a transistor. After crashing into the power wall, the designing approach shifted from employing a power-inefficient processor to usage of many efficient ones on the same chip, and hence promoting parallel computing rather than sequential programming [43]. While parallel computing is not new, during the recent years the interest in it has increased due to multi-cores becoming the norm in computing systems, ranging from smaller devices such as phones, to high-end servers. Therefore, it is true to say that the interest in parallel computing nowadays, is not only the result of an innovation in programming but also the actual limitations in building power-efficient, high clock-rate, single core chips [44].

A processor on a chip hosting multiple computational units (i.e. *cores*), is referred to as multicore processor. A system containing one or more multicore processors is referred as multicore system. Such systems enable parallelism (i.e. multi-threaded executions) by running tasks on different cores. Ideally, a large task is divided into several small independent sub-tasks, each is assigned to a core to be completed. However, distributing sub-tasks over cores is not always

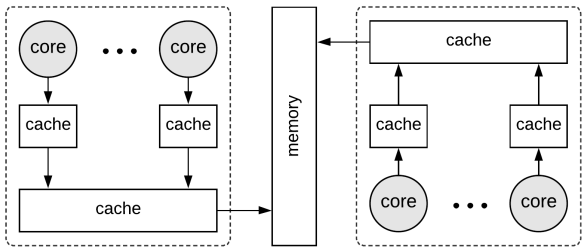

Figure 3: Classical architecture for shared-memory multicore systems

trivial. In many cases, sub-tasks have correlations that require the overlapping execution of *threads* to be synchronized through the shared memory.

Figure 3 illustrates a classical architecture for shared-memory multicore systems including several cores, connected via a shared memory, where each core may have several private and shared caches. The cores execute tasks independently and coordinate via a shared address space to which each core has access with different layer of memory hierarchy. In parallel computing, concurrent access of cores to a shared data object, needs to be synchronized in order to guarantee data consistency. This is provided by several *synchronization* mechanisms (e.g. locks, semaphores) and hardware primitives (e.g. compare-and-swap, test-and-set) using which, the notion of *atomic operations* is supported. Atomic operations are the ones that appear to be completed by a thread without any interference [45], [46].

Shared-Memory vs Shared-Nothing Architecture
In the concept of parallel processing, there are two predominant architectures, namely *shared-memory* and *shared-nothing*. In a shared-memory (or simply shared) environment, all processes (e.g. threads, computing nodes) have access to a shared pool of memory resources which implies the need for an appropriate synchronization among processes. In a shared-nothing environment, however, each process operates independently, and controls its own memory resources. In shared-nothing, data is partitioned among the processes, and the workload is distributed such that each process operates on its own data, without sharing hardware resources with other processes. In shared environments, the focus is on maximizing resource utilization while shared-nothing enhances independence and parallelism among threads by preventing contention over the shared resources.

2.3 Processing Models

In the literature, various big data processing approaches are introduced for different IoT applications that employ one or more of the aforementioned scaling techniques [15]. These approaches can be categorized to two common models, *batch processing* and *stream processing*.

2.3.1 Batch Processing

Traditional database management systems (DBMSs) have been used for decades to manage data. The primary goal of DBMSs is to store data in a form of persistent dataset and then run one-time queries over it. This is called *batch processing* model. One of the most famous and powerful batch processing tools for Big Data is Apache Hadoop which implements the computational paradigm named *MapReduce* [47]. MapReduce facilitates processing by splitting the huge amount of data into smaller chunks and running the analysis in parallel on Hadoop commodity servers. In the end, it aggregates all the data from multiple servers to return a consolidated output back to the application.

Although batch processing is great for many applications, it is not a real-time and high performance processing model. Therefore, for certain stream data applications in the field of IoT and CPS — which are relevant to the context of this thesis — stream processing for real-time analytics is mightily necessary.

2.3.2 Stream Processing

For many modern high-velocity data-driven applications where Big Data is being generated continuously, batch processing model, in which the data is first stored and then queried, is inefficient. The main reason of this inefficiency is the high cost of the frequent access to the storage in such model. As a response to this problem, one option is to remove the requirement of storing data before processing, and therefore analyse data upon receiving it. This modification has emerged a new paradigm for continuous processing, which is referred as *data stream processing*.

Stream processing is one pass analysis over the data on the fly. In contrast to batch processing, stream processing employs *continuous queries* [48] which are queries that are issued once and continuously run over the flow of data. As an example, consider a scenario where a sensor is used to monitor speed of a vehicle and raise an alert if it exceeds a certain threshold. Using stream processing, the continuous query is run over arriving data records (referred as *tuples* in the remaining), monitoring the speed of the vehicle. Figure 4 illustrates a high level overview of information processing procedure using batch and stream processing. As shown in the figure, stream processing can run the query immediately after a new tuple arrives which in turn enables online real-time analysis. However, it faces issues in the way that tuples need to be processed.

A challenging issue regarding data stream processing lies in the fact that naturally, streaming data is unbounded. In addition, since stream processing does not store data, there is a requirement to keep a portion of it to run the queries that need past data. In the vehicle example, assume we are interested in finding the average speed during the past hour and raise an alert if the average is above a threshold. To reason about time in stream processing, a tuple carries its *event-time*, among all other attributes according to the schema. Event time, which we refer to as *timestamp*, is the time at which events actually occur [49].

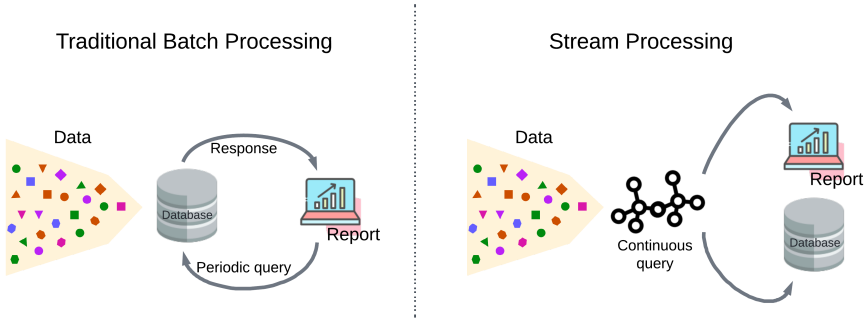

Figure 4: Illustration of information processing using traditional batch processing and stream processing.

Various models have been proposed to keep a portion of a stream of data with differences in downgrading the importance of the older tuples [50]. Among all, *sliding window* [51] is one of the prominent models in which only the most recent tuples that fit in a window are kept. The window itself can be either *time-based* or *tuple-based* and is defined by two parameters *size* and *advance*. In time-based windows, size is indicating the length of a window in time units and advance shows how much in time the window should go forward (e.g. to group latest tuples within a window of size 10 seconds every 3 seconds). In tuple-based windows, size is the length of a window in the order of number of tuples and advance indicates how many tuples the window should go forward (e.g. to group last 10 tuples every 3 receiving tuples).

To support the stream processing paradigm, *Stream Processing Engines* (SPEs) have been introduced as a new class of system software. Examples of such systems are STREAM [52], Apache Storm [53], Apache Flink [54], StreamCloud [55]. SPEs are generally providing high level programming interfaces to run continuous queries. The queries are modeled as Directed Graphs (DGs) where vertices are processing *operators* and directed edges are continuous streams of data. An operator is the basic processing unit in stream processing, receiving one or more streams, processing the data items in an online manner and producing one or more output streams. Special input operators (i.e. *Sources* or *Spouts*), deliver the streams to the streaming query. Moreover, the final results are handled by special output operators (i.e. *Sinks*). One of the most basic stream processing applications is a collection of three operators: a source operator, one of the processing operators, and a sink operator. The processing operators are either *stateless*, i.e. do not keep state as the result of previous tuples and perform actions based on each tuple individually (e.g. filter out the tuples using their attributes), or *stateful*, i.e. use state affected by the previous tuples to produce results (e.g. aggregate to compute average). Due to the unbounded nature of the streams, stateful operators are computed over windows.

Performance Metrics
Typically, the performance of SPEs is measured by two QoS metrics; *throughput* and *latency*. Throughput indicates number of tuples processed per time unit while latency is the timestamp differences of the output tuple and the latest input tuple that contributed to it. In stream processing, when the velocity of the arriving data exceeds the operator's maximum processing speed, the QoS of the whole streaming query might start to degrade. More specifically, since the operator will not be able to process data quickly enough, its input queue(s) will start growing in size. This, in turn, leads to an increase in operator's processing latency. Consequently, this latency will affect downstream operators, potentially leading to an increase in the overall latency of the streaming query. In such circumstances, one way to keep the latency low is to use a backpressure mechanism [56] and propagate the overload notifications from the operators backward to the data sources. By doing so, however, the throughput of the whole system might decrease.

Parallelism
In order to achieve higher throughput and lower latency, SPEs provide *task parallelism*, naturally, by assigning independent operators to different processing units. *Pipeline parallelism* is a subset of task parallelism, where the parallel operators have a producer-consumer relationship. Moreover, SPEs enable *data parallelism* by running multiple identical instances of a single operator on different subsets of the data in parallel. In data parallelism, a *splitter* partitions the input data, which is then processed by multiple identical parallel operator instances. The partitioning of the data stream for stateless operators can be accomplished in an arbitrary fashion (e.g. random, round-robin) while stateful operators require techniques that are aware of states and route tuples, that affect the same state, to the same partition. Finally, the parallel input streams are merged into a single stream by a merger. Nevertheless, the asynchronous and distributed execution of parallel instances can result in arbitrary interleaving of tuples from distinct sub-streams. This may cause that operators can be fed out-of-timestamp-order tuples [57]. In the following we review how to handle such out-of-timestamp-order streams.

Event-Time Ordering and Determinism
When dealing with out-of-timestamp-order tuples, one approach is the time-agnostic processing [49]. This approach is used for cases in which time is essentially irrelevant. A stateless operator (e.g. filter) is very basic form of time-agnostic processing. Since the stateless operator processes one single tuple at any time, having the input stream unbounded and unordered is not important. For stateful operators, however, the requirements are different. Considering the vehicle example, to compute the average speed of the vehicle for the past hour, the operator requires all tuples that belong exactly to the window covering the last hour to correctly produce the result. Therefore, a stateful operator needs to determine if it has received all the data that belongs to a certain window covering a portion of event-time. This can be done by enforcing event-time ordering across the whole query when the stream generated

by each data source or parallel instance of an operator is timestamp-sorted. To do so, one option is to merge the input streams of the stateful operator into one timestamp-sorted stream and then, process tuples from the latter in the timestamp order [58]. The most common benefit of enforcing event-time ordering is having deterministic executions [57].

When leveraging parallelism, deterministic execution should ensure that the results given by the parallel query are exactly the same that would be given by its centralized counterpart. *Determinism* in SPEs can be guaranteed by having (i) deterministic processing components, and (ii) deterministic flow of the results to downstream operators [59]. To ensure (i), it is important that the splitter, partitions the input stream(s) according to the semantics of the operation in a way that each input sub-stream contains all data required for the operation. Moreover, to ensure (ii), one way is to enforce event-time ordering and merge output sub-streams into one timestamp sorted stream. In Section 2.4, we discuss a concurrent data structure, ScaleGate — also leveraged in this thesis — that enables deterministic execution in stream processing.

2.4 Concurrent Data Structures

Shared data objects can be described through *Abstract Data Types* (ADTs). ADT is an interface definition of operations that can be executed on the data structure. An algorithmic implementation of ADT in the shared-memory system is a *concurrent data structure* which organizes data for efficient concurrent access while hiding details on the interaction of the processes. An efficient concurrent data structure is a key to harness the available parallelism in multicore systems by providing correct synchronization with high-level of interface operations. Design and implementation of concurrent data structures, nonetheless, are challenging as they are required to be consistent and scalable [45]. Such requirements are proved through the *safety* and *liveness* properties [60]. The safety property describes the consistency guarantees by showing that "something bad will not happen", while the liveness property describes the progress guarantees by showing "something good will eventually happen".

Various formalizations are presented in the literature for the safety property such as *linearizability* [61] and *sequential consistency* [62]. Linearizability preserves real-time occurrence of the operations and requires that each operation takes effect instantaneously at some point (i.e. linearization point) between its invocation and response. An execution is linearizable if there exists an ordered sequence of invocation and response events that observes real-time ordering of the latter at all processes. However, in some cases, the real-time order of events at different processes may not be significant. In such situations, sequential consistency is used as the safety condition. Instead of real-time order of events, sequential consistency preserves the program order of operations issued by the same process. Sequential consistency is a weaker condition compared to the linearizability since every linearizable execution also provides sequential consistency but the reverse is not necessarily true.

Similarly, the liveness property is defined through various formalizations

such as *wait-freedom* [63] and *lock-freedom* [45]. Wait-freedom is the strongest progress guarantee which concerns individual progress. It guarantees that *every* process has a bound on the number of steps to take before its operation completes regardless of delays or failures of other processes. Lock-freedom guarantees *some* process complete its operation after a bounded number of steps, and hence ensures the system-wide progress. During the past years, extensive effort has been made to construct more efficient and practical data structures (cf. [45], [46], [64], [65] and references therein). In the following, we review two state-of-the-art concurrent data structures, STINGER and ScaleGate, which are also leveraged in this thesis.

STINGER Data Structure

Spatio-Temporal Interaction Networks and Graphs Extensible Representation, STINGER, is a high performance concurrent data structure for streaming graph processing [66]. It is a shared memory data structure based on adjacency lists. STINGER's model consists of a vertex table and an edge list. Each element of the vertex table (called Logical Vertex Array) points to a given location in the edge list (Edge Block Array). The vertex table holds the vertices in the graph while the edge list consists of edge blocks, which hold the edges associated with each vertex. Edge blocks can point to other edge blocks to accommodate more space for edges belonging to a vertex.

The STINGER data structure permits the operations of query, insert, and delete, on both vertices and edges. Asynchronous processes may concurrently insert and delete vertices and edges from the STINGER data structure. In case of concurrency, timestamps are used to avoid conflicts on any insertion and deletion. For instance, when inserting an edge into a vertex's adjacency list, there are three possible scenarios; (1) if the edge already exists, the insert function should increment the edge weight and update the timestamp, (2) if the edge does not exist, a new edge should be inserted in the first empty space in an edge block of the appropriate type, and (3) if there are no empty spaces, a new edge block containing the new edge should be allocated and added to the list. The parallel implementation guarantees these outcomes by following a simple protocol using atomic compare-and-swap instructions.

In STINGER, parallelism exists at many levels. Each vertex has its own linked list of edge blocks that is accessed from the logical vertex array. Within an edge block, the incident edges can be explored in a parallel loop. The size of the edge block, and therefore the quantity of parallel work to be done, is a user-defined parameter.

ScaleGate Data Structure

ScaleGate [58], [67], [68] is concurrent data structure that is used to merge-sort the parallel streams. It efficiently merges several timestamp-sorted streams into one and allows the operator instances to process tuples in timestamp order once they are "ready". A tuple is defined as *ready* to be processed, if its timestamp is less than or equal to the latest tuple timestamps received from all input streams [58]. ScaleGate provides an encapsulated object which

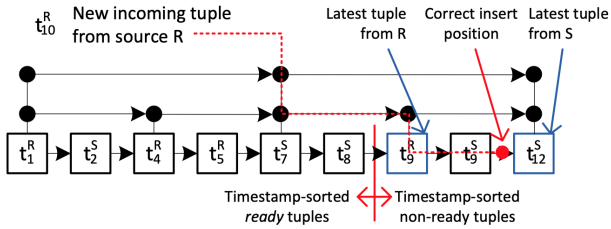

Figure 5: ScaleGate merge-sorting tuples from two input streams, R and S (adopted from [58]).

(i) guarantees properties essential for concurrently merging streams, (ii) integrates the necessary synchronization to allow multiple threads to consume ready tuples concurrently, and (iii) allows for an arbitrary number of source threads to deliver the tuples concurrently. The interface of ScaleGate provides the following methods:

- `addTuple(tuple,sourceID)`: which allows a `tuple` from the source thread `sourceID` to be merged by ScaleGate in the resulting timestamp-sorted stream of ready tuples.
- `getNextReadyTuple(readerID)`: which provides to the calling reader thread `readerID` the next earliest ready tuple that has not been yet consumed by the former.

The implementation of ScaleGate builds upon the idea of lock-free linearizable skip lists. In particular, the `addTuple` method implementation allocates a new node, searches for the appropriate position in the list, and tries to insert it in a lock-free manner. For updating the related pointers, an atomic compare-and-swap instruction is used. When retrieving tuples with the `getNextReadyTuple` method, a tuple will be returned only if it is ready, i.e. if it is not the last inserted tuple from the respective input source. This is checked by using a pointer back to the input source handle and comparing with the address of its last inserted node. Figure 5 shows an example structure of ScaleGate to merge-sort two input streams R and S into one timestamp sorted stream of ready tuples. In the figure, the dashed red line shows the search path for the insertion of a new tuple from stream R.

3 Research Problems

As mentioned earlier, continuous data processing is a general approach to analyze Big Data in the context of CPS. In the previous section, we provide background on topics related to machine learning, parallel computing, and stream processing. In this section, we describe problems and challenges associated with efficient continuous analysis divided into two parts, namely continuous data mining algorithms, and utilization of computing resources.

3.1 Continuous Data Mining Algorithms

As discussed before, continuous processing has several advantages over batch processing which make the former be more beneficial for CPS related applications. However, continuous processing is more complicated to design and implement as the result of the new challenges introduced by it [50]. In the following, we review a few of these challenges that are mainly caused by the single pass over the streams of data.

- **Accuracy**: The main challenge when designing a continuous processing algorithm is to achieve accurate results. Unlike conventional algorithms which have a complete knowledge about data by accessing the stored data probably multiple times, in continuous processing there is a complete lack of information regarding the upcoming data. In these circumstances, producing accurate results despite of incomplete information is a major challenge in designing continuous processing algorithms.

- **Data Management**: Another challenge that arises naturally when designing an algorithm for continuous processing is handling the continuously arriving tuples in an efficient manner. To do so, it is important to employ efficient data structures that allow frequent updates. Having such data structures is especially critical in case of parallelism where the concurrent access to the data by several processes requires some synchronizations to guarantee safety and liveness properties.

- **Performance**: To achieve high throughput of results while keeping the latency low, thus contribute to real-time analysis, it is necessary to save time and redundant work. This is particularly important when yielding results in multiple and configurable time granularity is desired which would cause higher latency. To this end, a challenging aspect in designing continuous processing algorithms is to incrementally update the results by reusing the previous computations.

3.1.1 Representative Examples for Continuous ML Applications

The design of algorithmic approaches that can enable continuous processing is problem dependent and hence appropriate analysis methods are required. In this section, we describe two case studies in the context of CPS, for which continuous data mining and ML could have large impacts. For each application, we discuss the aforementioned challenges in more details.

Case Study 1 - Spatio-Temporal Data Clustering
For the first application, we consider the problem of online clustering streams of spatio-temporal data while the latter is being received. To give a more intuitive description, we focus on LiDAR which generates data with similar characteristics.

LiDAR, which stands for Light Detection And Ranging, is a sensing technology to measure distance and depth data of objects. The idea behind LiDAR has been around since 1960s [69]. Nevertheless, it was only the recent years

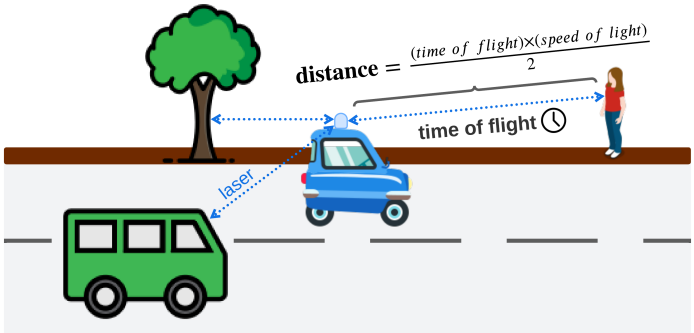

Figure 6: Illustration of a LiDAR system to show how it computes distances of the surrounding objects.

that LiDAR's capabilities have really opened up. Nowadays, LiDAR is used in different applications such as airborne, mobile phones, and smart transportation [70]–[72] to identify 3D shapes and objects. LiDAR technology is especially beneficial for autonomous vehicles as it is less influenced by light compared to cameras and consequently could provide more robust features in challenging environments. In such applications, a rotating LiDAR sensor (shortly LiDAR in the remaining) is mounted on a vehicle or a robot to provide 360 degrees horizontal field of view and detect surrounding objects such as other vehicles, pedestrians, cyclists, etc.

A typical LiDAR mounts several lasers on a column with different vertical shooting angles, each emitting pulsed light waves into the environment. These pulses bounce back to the sensor once hitting an object. LiDAR, then, calculates the distance of the reflected point by considering the time it took for the pulse to return to the sensor. Figure 6 illustrates a LiDAR system. The column of lasers also rotates horizontally very rapidly while producing distance readings which provides a real-time 3D map of the environment. The collection of 3D LiDAR distance readings is called *point cloud*.

A common way of extracting valuable information from LiDAR data (e.g. to detect objects surrounding the sensor) is by clustering [73]. As described in Section 2.1, distance based clustering approaches form clusters using a given distance metric. Since computing the distances between all pairs of n tuples to find the ones within the threshold would imply $O(n^2)$ operations, it is important to prune the search space. For this purpose, several clustering approaches introduce an intermediate step before performing the clustering algorithm. This additional step, organizes the tuples in a supporting data structure (e.g. kd-tree [31]) to speed up finding nearest neighbours. This way, a batch processing approach is introduced which improves the expected complexity to $O(nlogn)$ (i.e. not worst-case). However, such batch processing requires multiple passes over the data which in turn, might affect the performance.

LiDAR generates the data in streams which are implicitly ordered. Therefore, continuous clustering is desirable to produce streams of results while

organizing the data points in an additional step may be avoided. For instance, Klasing [74] leverages the inherent ordering of LiDAR data points. However, it achieves lower accuracy than the Euclidean distance based method. Moreover, Zermas et al. [75] proposed a clustering method specific to the structure of LiDAR data points. Though, it still relies on a kd-tree for some necessary nearest neighbour searches. In another work, Yin et al. [76] speed up clustering by grouping separate objects in the same azimuth zone using spherical coordinates with the cost of losing accuracy.

The aforementioned common state-of-the-art approaches indicate that achieving accurate results while performing the clustering continuously and efficiently poses challenges such as modifying the clusters while new points arrive. Besides, high data rates and the implied need for parallelization require efficient data structures for concurrent and frequent modifications. Moreover, considering that consecutive 360-degree scans of LiDAR most often have large similarities, results from previous rotations should be reusable, to prevent redundant computations and support incremental clustering. Furthermore, it is desired to yield the results in small time granularity for making further decisions quickly and attentively.

Case Study 2 - Bottleneck Monitoring in Industrial Processes

For the second application, we consider the problem of continuous monitoring to detect bottlenecks in industrial processes. A system concept that enables monitoring and collecting data from manufacturing activities across globally distributed plants is Manufacturing Execution Systems (MES). The MES concept was born from the demand on the manufacturing enterprise to fulfil the requirements of markets from a point of view of reactivity, quality, and reduction in cost. In principle, MES provides the log of the machines' activities from the shop floor in real time to improve the functionality and resource utilization of the system as well as to react to risky situations [77].

By analysing the data provided using MES, bottlenecks can be monitored on a real-time basis in order to reduce the response time and implement improvement actions, thereby facilitate real-time production [35]. However, most of the current bottleneck detection methods are simulation-based approaches which are time consuming [78]. Such approaches also require huge efforts in terms of developing the simulation model of the production system and updating the simulation model with the changes made in the real production system. Moreover, since simulation-based approaches take a long time to run, it is difficult to obtain results on a real-time basis [79]. To improve simulation-based approaches, several data driven methods are presented in literature. For instance, the turning point method [79], which is a simulation based approach, uses the blockage and the starvation data of machines to detect the bottlenecks. However, the online measurements of blockage and starvation times of the machines are greatly affected by the buffers in the production system and therefore fail to detect the true bottlenecks in a completely decoupled system with large buffers and frequent small stoppages in the machines. Another approach that facilitates the bottleneck detection procedure is the shifting bottleneck identification [78]. Nevertheless, the method needs to be further

developed to yield results continuously.

The aforementioned approaches indicate important challenges for real-time monitoring such as detecting bottlenecks in an efficient and timely manner while the accuracy of the results is not traded for the performance. Another important challenge is to detect the bottlenecks at configurable time granularity (e.g. the bottlenecks of minutes, hours, shifts, days, months, etc), since the latter can provide great insight for improvements in the production pipelines. It is also critical to employ data structures that not only enable frequent modifications efficiently, but also allow timely extraction of the results.

3.2 Utilization of Computing Resources

As mentioned before, the major challenge of CPS applications is given by the volume and velocity of the data that is produced continuously. For instance, a modern vehicle senses approximately 4 terabytes of data a day [80] containing huge amount of information. Unlocking the potential of this information hinges on the development of sophisticated analytical frameworks and scalable tools capable of exploiting all computational resources in cost-effective ways. Depending on the analytics activity, the specific requirements include, among others, the distribution of the processing procedure across multiple tiers and elastic resource acquisition. In the following we discuss these two requirements, as the focus of this thesis, in more details.

3.2.1 Multi-tier Processing

Given the current data growth, having an analysis that pipelines data generation and processing is a challenging necessity. Therefore, as mentioned before, processing is no longer done only in massive servers but distributed across a wide spectrum, including intermediate processing layers (fog nodes) and resource-constrained devices (edge devices). Moreover, by distributing the processing tasks on multiple tiers, the communication bandwidth will no longer be a limiting factor.

Nevertheless, due to different characteristics of the resources deployed at various tiers, the concept of multi-tier computation implies some trade-offs. At one end of the spectrum, clouds typically consist of high-end servers with abundant storage and massive computing resources which enable high parallel computing. Therefore, performing analysis on data-intensive applications in the cloud would decrease the processing latency. However, if the accuracy is a top priority for the application (which implies that all data needs to be sent to the cloud), processing all data in the cloud would increase the load as well as the latency of the communication medium. At the other end, the computational and memory of the edge devices are very limited. This means, pushing the computations to the lower tiers, although decreases the communication latency, might increase the processing latency. Moreover, distributing the computation on the edge devices might require combining the partial results produced by these devices, which in turn, introduces an extra latency.

In these circumstance, it is important to know which type of processing is

best done in which part of the analysis pipeline. Generally, existing literature scale down the collected data on the edge, and then, transfer the reduced data to the upper tiers to run the heavy analysis. For instance, data sanitization (i.e. pre-processing) is applied on the data before performing different analysis [81]. Here, the sanitization usually includes operations such as removing information outside the time interval of interest, extreme outliers, and faulty signals. Moreover, various solutions in the literature (e.g. [41], [82]–[84]) use approximation techniques to reduce the size of the data by compacting the latter. Each of these solutions focuses on different aspects of the approximation (bounded error, processing time, etc.). A recent work [85] targets at compressing wireless sensor streams before transmission. In this work, PLA (discussed in Section 2.2.1) is used on embedded edge devices to reduce the size of the data, thus, reduce communication and energy consumption. In another recent work [86], the authors propose a PLA algorithm to be run on resource-constrained wireless sensor nodes.

While the aforementioned approaches indicate the necessity of utilizing the cumulative computational power of edge devices, they do not discuss the effects of factors which cause out-of-timestamp order data such as varying speeds and reliability of the underlying communication layer. Moreover, when compacting the data to scale it down at the edge, an important challenge that needs more investigation is the trade-off between space saving and accuracy loss. Furthermore, in addition to distributing the computation on the edge devices, it is important to utilize the resources over the computing continuum in order to achieve high throughput of results with low latency.

3.2.2 Elasticity

In the concept of stream processing, there is rich scientific literature on leveraging the modern multicore systems' computational power in terms of parallelism where the number of parallel instances of an operator is fixed. Such techniques focus on a specific operator parallelization [58], [87], [88] or determinism [58], [59], among other aspects. These fixed resources, nonetheless, are not able to be adjusted at runtime and therefore render over- or under-provisioning scenarios in the case of changing stream rates [89].

In the over-provisioning scenario, the number of allocated resources is set regarding the load peaks, which causes under-utilization of the system and high cost during non-peak times. In the under-provisioning scenario, there are not enough resources, which causes higher analysis latency and, in the case of bursty workload, saturation in the system [90]. Moreover, when the overall work is unbalanced but could be carried out by the available threads as a whole, a load balancing adaptive reconfiguration is needed to change the work distribution to threads. Therefore, *elasticity* is an important requirement that has to be provided in order to enhance resource utilization in the system.

The elasticity mechanism enables runtime resource adjustments to adapt the number of parallel computational instances regarding the stream rate. It introduces challenges especially for the stateful analysis where redistribution of work among new number of resources is often followed by *state migration* [90],

[91]. State migration is the procedure of transferring the current state from one thread to the new one, as the latter has been running the analysis.

As an example, consider a user that is interested in parallelizing the analysis of a streaming application that consumes tweets and runs analysis on a per-hashtag basis. The user could leverage multiple threads to run the analysis in parallel by assigning distinct hashtags to such threads. This, nonetheless, cannot be done in a straightforward fashion, since tweets can carry two or more hashtags assigned to different threads. A possible approach is to copy each tweet (as many times as the hashtags each carries in the worst case). However, the cost of data duplication hampers unnecessarily the performance. The second type of overhead is caused by the operators' dedicated internal state. Since instances' states are not shared, state transfer is needed in elastic reconfigurations to adjust the workload distribution and/or parallelism degree of an operator.

While these overheads are unavoidable for distributed shared-nothing systems, they are not for parallel instances that share memory, and could thus share tuples and states too. In addition, as discussed in Section 2.2, to take advantage of all available resources, it is required to efficiently manage and scale computation on a node level first (i.e. scaling up) before scaling out to the distributed nodes. Another challenge in programming elastic stateful analysis, relies on preserving determinism even with varying degree of parallelism which, in turn, might require elastic and re-configurable data structures.

4 Thesis Objectives

In this thesis, we investigate aspects of continuous processing, with a focus on CPS applications, to achieve high throughput with low latency while preserving determinism by designing, implementing, and evaluating efficient analytics; i.e., algorithms that perform timely processing on streams of data, and frameworks that can improve utilization of resources and balance application performance metrics. The thesis particularly concentrates on the following research questions:

RQ1 How are throughput, latency, and precision impacted by continuous processing streams of data?

RQ2 How can efficient data structures for scalable and configurable continuous analysis, contribute to real-time processing?

RQ3 How do pipelining the analysis and utilization of multiple tiers of the computing continuum affect the analysis speed and costs?

RQ4 How can we achieve instantaneous elastic reconfigurations in stream processing while preserving determinism?

RQ1 is relevant to the applications for which continuous processing is more beneficial than traditional batch processing, due to the high rate of the incoming data and the demand for online processing. Such applications, typically, require

Table 1: Research questions addressed in each chapter.

	Chapter A	*Chapter B*	*Chapter C*	*Chapter D*	*Chapter E*
RQ1	●	●	●	●	○
RQ2	●	●	●	●	●
RQ3	○	○	●	●	○
RQ4	○	○	○	○	●

high throughput of results with low latency. To this end, online processing of streams of data raises interesting research questions on how to analyse data in one pass and achieve high performance without losing accuracy of results.

RQ2 emphasizes the need for efficient data structures to manage the flow of data in continuous processing and match the rate of processing with that of incoming data. This is especially important in case of having parallel computing in order to enable safe and efficient concurrent accesses for parallel processes. Today, suitable hardware for parallel processing can be found on all tiers of the computing continuum which indicates the necessity of having scalable algorithms and efficient concurrent data structures.

RQ3 is relevant in the context of moving processing away from the cloud and distributing it on multiple tiers. By doing part of the processing on the edge, while still performing heavy analysis on the upper tiers, the bandwidth between different tiers will be utilized as the result of the scaled down data. However, this might cause data loss and affect the accuracy of the results. To this end, enabling multi-tier data processing is a challenging research topic which entails a practical trade-off between data reduction and pipeline speed on the one hand, and accuracy loss on the other.

RQ4 becomes particularly important by considering the varying rate of the incoming data that needs to be processed. Unpredictable behaviour of the data rate makes it even more difficult to prepare the system. Therefore, it is important to have frameworks which adjust the processing resources with respect to the incoming rate. This raises several research questions on how to provide a framework that allows reconfiguration during the runtime with deterministic execution to maximize efficiency in terms of throughput, latency and reconfiguration times.

We relate back to these research questions and how we address them in the context of the thesis contributions discussed next.

5 Thesis Contributions

The contributions of this thesis are solutions for a number of data analytical challenges on designing continuous data mining/ML algorithms and utilizing computing resources, mentioned in Section 3. In this section, we outline the results and connect to the research questions raised in Section 4. The structure of the following chapters of this thesis and the research questions they address are described in Table 1.

5.1 Continuous Data Mining Algorithms

In the first part, we target continuous data mining and ML algorithms through two representative problems: (i) spatio-temporal data clustering, and (ii) bottleneck monitoring. The common challenge that we address in both applications is to provide a single pass analysis of streams of data in an efficient and timely manner while producing accurate results. By doing so, we contribute towards RQ1 and RQ2 and show (i) how continuous processing of streams of data can produce accurate results even with incomplete information, and (ii) how efficient data structures can benefit continuous processing.

5.1.1 Spatio-Temporal Data Clustering

Chapter A

We begin this problem in Chapter A by focusing on LiDAR point cloud clustering, since LiDAR allows more intuitive description of data with spatio-temporal locality. For that, we introduce *Lisco*, a single-pass continuous Euclidean distance based clustering that can enable fine-grained pipelining and parallelism. The intuition behind *Lisco* is to leverage the sorted delivering of tuples from the LiDAR. Therefore, unlike the state-of-the-art Euclidean distance based clustering [31] that organises data in a supporting data structure to find ϵ-neighbours of point p and cluster them together, *Lisco* translates the ϵ-neighbourhood of p into a set of readings given by certain steps and lasers around p, thus, eliminates the need for a search-optimized data structure. This way, while points are being collected, *Lisco* performs clustering on them efficiently. Subsequently, we propose the parallel version of *Lisco*, *p-Lisco*, which is architecture-independent and scales the analysis by exploiting the inherent disjoint-access parallelism of the algorithm's pipeline and of the calculations needed for each point.

Lisco and *p-Lisco* address the challenge of incomplete information by leveraging the symmetric characteristic of the Euclidean distance. Considering this characteristic, the algorithms guarantee if two points have a distance less than ϵ, upon processing the point that arrives second, the two points will be grouped together. The contributions of *p-Lisco* also include a concurrent efficient data structure which enables continuous data management as well as orchestrating all parallel threads to avoid contention. Moreover, the lock-free, linearizable implementation of the proposed shared data structure in *p-Lisco* allows balanced partitioning of the work among parallel threads. By addressing the issue of incomplete information and managing the continuous data as well as the intermediate results efficiently, the proposed algorithms provide accurate clustering results at the end of each rotation.

We analyze the complexity behaviour of *Lisco* and compare it with that of state-of-the-art. We show that $O(nlogn)$ is *Lisco*'s time complexity in the worst case, whereas it is the expected complexity for state-of-the-art. Furthermore, we present a thorough comparative evaluation, using both real-world and synthetic data. We use hardware representative of devices in the computing continuum and observe that *Lisco* offers high benefits already in its sequential from. It is

also shown that, in cases with more workload, when it is possible to employ higher number of threads, *p-Lisco* shows a significant improvement compared to the baseline.

Chapter B

We continue the problem of LiDAR point cloud clustering by advancing *Lisco* algorithm to re-use computations between rotations. For that, we propose parallel and incremental *Lisco*, *pi-Lisco*, whose key idea is the exploitation of the locality properties of the points that belong to the same clusters, not only in the same LiDAR rotation, but also in subsequent rotations' point clouds. The algorithm *pi-Lisco* enables incremental processing through using an efficient data structure to manage the incoming data continuously and re-using the parts of the data structure which had not any updates between subsequent rotations. The algorithm also leverages parallelism by partitioning large part of the work and assign the latter to the parallel threads.

To ensure deterministic executions, *pi-Lisco* exploits the synchronization and data-sharing properties of the ScaleGate data structure (discussed in Section 2.4), that allows disjoint parallelism in a fine-grained manner. Moreover, the algorithm proposes a customised version of the STINGER data structure (discussed in Section 2.4) that facilitates further to concurrently and efficiently share spatial data. Furthermore, we discuss the consistency and work-saving properties of *pi-Lisco* and its algorithmic implementation as a streaming operator to output clusters in configurable regular intervals. Using real-world data set and an extensive evaluation, we show the computational benefits from incrementally processing the consecutive point clouds, as well as continuously increasing sustainable rates with increasing number of threads.

Chapter C

As a part of a work that have done in chapter C, we generalize *Lisco*, and propose *g-Lisco*, to cluster data with spatio-temporal locality. The algorithm *g-Lisco* presents a distance based clustering approach which exploits the implicit sorting carried by one or more attributes of the tuples to speed-up the clustering procedure. Similar to the previous versions, *g-Lisco* achieves accurate clustering results while leveraging efficient data structures for continuous data management. In this work, we show the benefits of employing *g-Lisco* for several use cases, such as clustering LiDAR data transmitted from a single vehicle, and clustering GPS positions transmitted from a fleet of vehicles.

5.1.2 Bottleneck Monitoring

Chapter D

For the problem of continuous monitoring and bottleneck detection in industrial processes, in a part of chapter D, we introduce an efficient continuous shifting bottleneck detection algorithm, *Amble*. Unlike the state-of-the-art shifting bottleneck detection method [36] that first collects the data in batches and then

performs the analysis, *Amble* provides a fine-grained continuous processing with a single pass over data while achieving accurate results.

The main idea of *Amble* is to use a tree data structure, which is updated frequently by pruning and inserting new tuples upon reception, thus, maintaining all chains of momentary bottlenecks. The chain of momentary bottlenecks in a production line specifies how several machines might affect the performance of each other. Using the tree data structure, *Amble* enables fast identification of the bottleneck chain at any time, by simply making the chain equivalent to a path in the tree.

We implement *Amble* as an operator in stream processing and analyze the complexity of *Amble* by distinguishing two parts (i) the complexity of node insertion in the tree, and (ii) the complexity of traversing the tree to announce the chain of bottlenecks. In a system with m machines, the insertion complexity is $O(m)$ while the traversing complexity is linear on the path representing the bottleneck chain in the tree. We also evaluated *Amble* with real data to show the benefits that the former provides by enabling timeliness, configurable, and continuous processing compared to the state-of-the-art.

5.2 Utilization of Computing Resources

In the second part, we target distributing the work over computing resources at (i) multiple tiers, and (ii) single node. Specifically, we focus on scaling down the data before distributing the work over the computing continuum, and scaling up the processing on a node level. Therefore, we contribute to RQ3 and RQ4 by studying the effect of different scaling down policies on the performance and enabling intra-node elastic reconfiguration to adjust the computing resources at runtime.

5.2.1 Multi-tier Processing

Chapter C

In this chapter, we target the problem of distributing the work over multiple tiers with a focus on the application of vehicular networks. Motivated by the limited communication bandwidth compared to the volume of sensed data in vehicular networks and the monetary costs of data transmission, we study the trade-off between scaling down the data and the accuracy lost.

To this end, we present a framework, called *DRIVEN*, to utilize computational resources and communication bandwidth of different tiers in the edge/fog/cloud architecture. The intuition behind *DRIVEN* is to avoid gathering the data to be processed in a raw format from each vehicle, but rather to allow for a configurable streaming-based error-bounded approximation, through PLA, to compress the volumes of data to be gathered. Moreover, *DRIVEN* leverages *g-Lisco* in upper tiers to cluster the compact data in a continuous fashion. This way, *DRIVEN* provides a framework that pipelines data generation and processing. Therefore, the latency of the components overlap instead of being additive. Using thorough evaluation with real data, we show *DRIVEN* allows tuning the trade off between the amount of data and the accuracy in

the form of approximation. We also discuss the benefits of using *g-Lisco* as a continuous processing approach in the *DRIVEN* framework.

Chapter D

Similar to Chapter C, this chapter also studies the problem of multi-tier computation and scaling down the data, but with a different focus. In this chapter, rather than compacting the raw data, we investigate the effect of running pre-processing and filtering on the data at the edge. To design appropriate problem-analysis methods, we consider an industrial process and the problem of monitoring throughput bottlenecks.

We provide a framework, *STRATUM*, to support configurable and automated analysis, leveraging stream processing and enhancing task parallelism by distributing the work on the embedded processing units at the machines, as well as the analysis center. In *STRATUM*, the lower tier is responsible for data validation and filtering, while the upper tier takes care of the combined data-stream analysis. For continuously monitoring bottlenecks, the *Amble* algorithm is used in the framework. Moreover, to support deterministic execution, we use a ScaleGate component in the framework to coordinate and sort all streams emitted from different machines.

We evaluate the potential of *STRATUM* by running an experiment with data collected from a production system over two years. The evaluation of *STRATUM*, together with *Amble*, indicates the capability of the proposed tools in accurately detecting bottlenecks with configurable time granularity. Moreover, we discuss that, since *STRATUM* pipelines collecting and validating data at the lower tier with the analysis using *Amble* at the upper tier, the latency of *Amble* is hidden in the pipeline. Therefore, the *STRATUM* framework contributes to online and real-time analysis.

5.2.2 Elasticity

Chapter E

In this chapter, we propose the notion of *virtual shared-nothing parallelism*, using which it is possible to define parallel and elastic SPE operators that, while virtualizing the common APIs based on shared-nothing parallelism, leveraging shared memory to scale streaming applications up, before scaling them out.

To leverage virtual shared-nothing parallelism, we introduce a framework, *STRETCH*, that allows exploitation of parallelization techniques and shared-memory synchronization to boost the scaling up. The framework offers instantaneous elastic reconfigurations, without state transfer, while supporting determinism even with varying degree of parallelism. Moreover, we provide formal guarantees and correctness proofs for the semantics of the analysis tasks supported by *STRETCH*, showing they extend the ones found in common SPEs. As means to apply the elasticity, we also extend the API for ScaleGate [58], and propose *Elastic ScaleGate* to enable thread provisioning and decommissioning at runtime. With lock-free, linearizable algorithmic implementation, Elastic ScaleGate supports efficient reconfigurations without breaking determinism.

We provide an extensive discussion and an exhaustive evaluation, based both on synthetic and real data. We also compare *STRETCH*'s performance with that of various (well established) baselines by studying different metrics such as throughput, latency, and reconfiguration time. For instance, it is shown in one of the experiments that, by enabling state transfer-free elasticity, *STRETCH* offers unprecedented ultra-fast reconfigurations, taking less than 40 milliseconds even when provisioning tens of new operator instances.

6 Conclusions

In this thesis, we show that timely continuous processing is crucial in this era of Big Data where data is being generated continuously with high rate. We demonstrate this by addressing several challenges (e.g. performance, precision, determinism, etc.) for designing continuous data mining and ML algorithms as well as utilizing resources over the computing continuum.

We first focus on the challenges associated with continuous data mining and ML. We start by proposing three algorithms, *Lisco*, *p-Lisco*, and *pi-Lisco* to continuously cluster LiDAR data points in sequential, parallel, and incremental manner, respectively. *Lisco* is a single pass Euclidean distance based clustering that maximizes the granularity of the data processing pipeline and thus shows the potential for parallelism. *p-Lisco* exploits the parallelism of *Lisco*'s processing pipeline in an architecture-independent fashion and thus, contributes to real-time processing. *pi-Lisco* extends the previous versions to allow for continuous, parallel and incremental clustering in order to benefit from the completed work in previous rotations and avoid unnecessary recalculations. Moreover, we introduce a generalized version of *Lisco*, *g-Lisco*, which continuously clusters data with spatio-temporal locality by exploiting the implicit sorting carried by the attributes of the data. Furthermore, we introduce *Amble*, a continuous monitoring algorithm to detect throughput bottlenecks of an industrial process in an efficient, automated and configurable manner.

Next, we focus on the challenges associated with distributing the work to utilize available resources at (i) multiple tiers, and (ii) single node. For multi-tier utilization, we develop two frameworks leveraging stream processing, *DRIVEN* and *STRATUM*, that scale down the raw data using the embedded devices at the edge, and then transfer the reduced data to upper tiers for further processing. While *DRIVEN* studies the effect of compacting the raw data on the accuracy of results and the communication bandwidth in order to enable real-time analysis, *STRATUM* studies the performance improvement, in terms of throughput and latency, when the raw data is pre-processed at the edge. Furthermore, to address the challenges of utilizing resources at a single node, we introduce *STRETCH*, a framework that provides virtual shared-nothing parallelism and supports determinism while easing the programming of intra-node scalable, elastic, high-throughput and low-latency stateful streaming analysis.

The achievements of *Lisco* algorithm in clustering LiDAR point clouds motivate investigating other applications with strict time constraints that can

benefit from continuous processing. One such application is online object tracking which is particularly interesting for security and safety reasons. For instance, a moving object in a prohibited area can be detected online to react properly, or vehicles can continuously monitor other moving objects for various purposes such as path planning and obstacle avoidance. It would be also interesting to investigate the opportunities brought by *pi-Lisco* to track an object in consecutive rotations of LiDAR point clouds.

As the field of IoT and Industry 4.0 grow, manufacturing processes are becoming increasingly digital, which indicates the demand for frameworks such as *DRIVEN* and *STRATUM*. In this sense, an important step forward would be to identify the benefits of the proposed frameworks in designing a digital twin (i.e. a digital image of a physical object or process that helps optimize business performance) for industrial automation and predictive maintenance. Furthermore, based on insights from the work in elasticity, there are interesting aspects worth to be investigated as followup work. An example would be combining scale-out capabilities with *STRETCH*'s scale-up mechanism so to have a framework which allows scaling up first and then scaling out if still needed. Another exciting direction for the future work is to provide automatic insightful control mechanisms for the *STRETCH* framework to decide on the reconfiguration, and for the *DRIVEN* framework to decide on PLA-error margins.

Bibliography

[1] M. Wollschlaeger, T. Sauter and J. Jasperneite, 'The future of industrial communication: Automation networks in the era of the Internet of Things and Industry 4.0,' *IEEE Industrial Electronics Magazine*, vol. 11, no. 1, pp. 17–27, 2017 (cit. on p. 3).

[2] N Jazdi, 'Cyber physical systems in the context of Industry 4.0,' in *IEEE International Conference on Automation, Quality and Testing, Robotics*, 2014, pp. 1–4 (cit. on pp. 3, 4).

[3] A. Whitmore, A. Agarwal and L. Da Xu, 'The Internet of Things - A survey of topics and trends,' *Information Systems Frontiers*, vol. 17, no. 2, pp. 261–274, 2015 (cit. on p. 3).

[4] L. Columbus, *Roundup of Internet of Things forecasts and market estimates*, `https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016`, 2016 (cit. on p. 3).

[5] S. LaValle, E. Lesser, R. Shockley *et al.*, 'Big Data, analytics and the path from insights to value,' *MIT Sloan Management Review*, vol. 52, no. 2, p. 21, 2011 (cit. on p. 3).

[6] J. Qiu, Q. Wu, G. Ding *et al.*, 'A survey of machine learning for Big Data processing,' *EURASIP Journal on Advances in Signal Processing*, vol. 2016, no. 1, pp. 1–16, 2016 (cit. on p. 3).

[7] X.-W. Chen and X. Lin, 'Big Data deep learning: Challenges and perspectives,' *IEEE access*, vol. 2, pp. 514–525, 2014 (cit. on p. 3).

[8] D. D. Sánchez-Gallegos, D. Carrizales-Espinoza, H. G. Reyes-Anastacio *et al.*, 'From the edge to the cloud: A continuous delivery and preparation model for processing big IoT data,' *Simulation Modelling Practice and Theory*, vol. 105, pp. 102–136, 2020 (cit. on p. 3).

[9] M. Stonebraker, U. Çetintemel and S. Zdonik, 'The 8 Requirements of real-time stream processing,' *SIGMOD Rec*, vol. 34, no. 4, 42–47, 2005 (cit. on p. 4).

[10] J. Han, J. Pei and M. Kamber, *Data mining: Concepts and techniques*. Elsevier, 2011 (cit. on pp. 4, 9).

[11] D. Laney, *3-D data management: Controlling data volume, velocity and variety*, `http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`, 2001 (cit. on p. 4).

[12] G. I. Glossary, `https://www.gartner.com/it-glossary/big-data/`, 2019 (cit. on p. 4).

[13] A. Gandomi and M. Haider, 'Beyond the hype: Big Data concepts, methods, and analytics,' *International Journal of Information Management*, vol. 35, no. 2, pp. 137–144, 2015 (cit. on p. 5).

[14] S. Sagiroglu and D. Sinanc, 'Big Data: A review,' in *International Conference on Collaboration Technologies and Systems (CTS)*, 2013, pp. 42–47 (cit. on p. 5).

[15] C. Philip Chen and C.-Y. Zhang, 'Data-intensive applications, challenges, techniques and technologies: A survey on Big Data,' *Information Sciences*, vol. 275, pp. 314–347, 2014 (cit. on pp. 5, 12).

[16] P. Tedeschi and S. Sciancalepore, 'Edge and fog computing in critical infrastructures: analysis, security threats, and research challenges,' in *IEEE European Symposium on Security and Privacy Workshops*, 2019, pp. 1–10 (cit. on p. 5).

[17] H. Li, K. Ota and M. Dong, 'Learning IoT in edge: deep learning for the Internet of Things with edge computing,' *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018 (cit. on p. 5).

[18] E. M. Tordera, X. Masip-Bruin, J. Garcia-Alminana *et al.*, 'What is a fog node? A tutorial on current concepts towards a common definition,' *arXiv preprint arXiv:1611.09193*, 2016 (cit. on p. 6).

[19] L. Zhou, S. Pan, J. Wang and A. V. Vasilakos, 'Machine learning on Big Data: Opportunities and challenges,' *Neurocomputing*, vol. 237, pp. 350–361, 2017 (cit. on p. 6).

[20] D. Kimovski, R. Mathá, J. Hammer *et al.*, 'Cloud, fog, or edge: Where to compute?' *IEEE Internet Computing*, vol. 25, no. 4, pp. 30–36, 2021 (cit. on p. 7).

[21] D. Roman, N. Nikolov, A. Soylu *et al.*, 'Big Data pipelines on the computing continuum: Ecosystem and use cases overview,' in *IEEE Symposium on Computers and Communications (ISCC)*, 2021, pp. 1–4 (cit. on p. 7).

[22] D. Balouek-Thomert, E. G. Renart, A. R. Zamani *et al.*, 'Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows,' *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1159–1174, 2019 (cit. on p. 7).

[23] J. Diaz-Rozo, C. Bielza and P. Larrañaga, 'Machine learning-based CPS for clustering high throughput machining cycle conditions,' *Procedia Manufacturing*, vol. 10, pp. 997–1008, 2017 (cit. on p. 8).

[24] C. S. Wickramasinghe, K. Amarasinghe, D. L. Marino *et al.*, 'Explainable unsupervised machine learning for cyber-physical systems,' *IEEE Access*, vol. 9, pp. 131 824–131 843, 2021 (cit. on p. 8).

[25] Z. Fu, M. Almgren, O. Landsiedel and M. Papatriantafilou, 'Online temporal-spatial analysis for detection of critical events in cyber-physical systems,' in *IEEE International Conference on Big Data*, 2014, pp. 129–134 (cit. on p. 8).

[26] D. J. Patterson, L. Liao, D. Fox and H. Kautz, 'Inferring high-level behavior from low-level sensors,' in *International Conference on Ubiquitous Computing*, Springer, 2003, pp. 73–89 (cit. on p. 8).

[27] A. K. Jain, 'Data clustering: 50 years beyond K-means,' *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010 (cit. on p. 9).

[28] J MacQueen, 'Classification and analysis of multivariate observations,' in *Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, 1967, pp. 281–297 (cit. on p. 9).

[29] 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 *International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231 (cit. on p. 9).

[30] M. Ankerst, M. M. Breunig, H.-P. Kriegel and J. Sander, 'OPTICS: Ordering points to identify the clustering structure,' in *ACM Sigmod record*, vol. 28, 1999, pp. 49–60 (cit. on p. 9).

[31] R. B. Rusu, N. Blodow, Z. C. Marton and M. Beetz, 'Close-range scene segmentation and reconstruction of 3D point cloud maps for mobile manipulation in domestic environments,' in *Intelligent Robots and Systems*, 2009, pp. 1–6 (cit. on pp. 9, 20, 26).

[32] E. M. Goldratt and J. Cox, *The goal: A process of ongoing improvement*. Routledge, 2016 (cit. on p. 9).

[33] C. Betterton and S. Silver, 'Detecting bottlenecks in serial production lines – a focus on interdeparture time variance,' *International Journal of Production Research*, vol. 50, no. 15, pp. 4158–4174, 2012 (cit. on p. 9).

[34] P. Faget, U. Eriksson and F Herrmann, 'Applying discrete event simulation and an automated bottleneck analysis as an aid to detect running production constraints,' in *Winter Simulation Conference*, 2005, pp. 1401–1407 (cit. on p. 9).

[35] C. Roser, M. Nakano and M Tanaka, 'A practical bottleneck detection method,' in *Winter Simulation Conference*, vol. 2, 2001, pp. 949–953 (cit. on pp. 9, 21).

[36] C. Roser, M. Nakano and M. Tanaka, 'Throughput sensitivity analysis using a single simulation,' in *Winter Simulation Conference*, vol. 2, 2002, pp. 1087–1094 (cit. on pp. 10, 27).

[37] P. B. Gibbons, 'Big Data: Scale down, scale up, scale out,' in *IPDPS*, 2015, p. 3 (cit. on p. 10).

[38] C. S. Liew, A. Abbas, P. P. Jayaraman *et al.*, 'Big Data reduction methods: a survey,' *Data Science and Engineering*, vol. 1, no. 4, pp. 265–284, 2016 (cit. on p. 10).

[39] U. Jayasankar, V. Thirumal and D. Ponnurangam, 'A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications,' *Journal of King Saud University-Computer and Information Sciences*, vol. 33, no. 2, pp. 119–140, 2021 (cit. on p. 11).

[40] D. Zordan, B. Martinez, I. Vilajosana and M. Rossi, 'On the performance of lossy compression schemes for energy constrained sensor networking,' *ACM Transactions on Sensor Networks (TOSN)*, vol. 11, no. 1, pp. 1–34, 2014 (cit. on p. 11).

[41] E. Keogh, S. Chu, D. Hart and M. Pazzani, 'An online algorithm for segmenting time series,' in *IEEE international conference on data mining*, 2001, pp. 289–296 (cit. on pp. 11, 23).

[42] R. Duvignau, V. Gulisano, M. Papatriantafilou and V. Savic, 'Streaming Piecewise Linear Approximation for efficient data management in edge computing,' in *ACM/SIGAPP Symposium On Applied Computing*, 2019, 593–596 (cit. on p. 11).

[43] S. Akhter and J. Roberts, *Multi-core programming*. Intel press Hillsboro, 2006, vol. 33 (cit. on p. 11).

[44] K. Asanovic, R. Bodik, J. Demmel *et al.*, 'A View of the Parallel Computing Landscape,' *Communications of the ACM*, vol. 52, no. 10, 56–67, 2009 (cit. on p. 11).

[45] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2011 (cit. on pp. 12, 16, 17).

[46] M. Herlihy, N. Shavit, V. Luchangco and M. Spear, *The art of multiprocessor programming*. Newnes, 2020 (cit. on pp. 12, 17).

[47] J. Dean and S. Ghemawat, 'MapReduce: simplified data processing on large clusters,' *Communications of the ACM*, vol. 51, no. 1, 107–113, 2008 (cit. on p. 13).

[48] S. Babu and J. Widom, 'Continuous queries over data streams,' *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001 (cit. on p. 13).

[49] T. Akidau, S. Chernyak and R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing*. " O'Reilly Media, Inc.", 2018 (cit. on pp. 13, 15).

[50] M. Garofalakis, J. Gehrke and R. Rastogi, *Data stream management: Processing high-speed Data streams*. Springer, 2016 (cit. on pp. 14, 19).

[51] M. Datar, A. Gionis, P. Indyk and R. Motwani, 'Maintaining stream statistics over sliding windows,' *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1794–1813, 2002 (cit. on p. 14).

[52] A. Arasu, B. Babcock, S. Babu *et al.*, 'STREAM: The Stanford stream data manager,' *IEEE Data Engineering Bulletin*, vol. 26, no. 1, pp. 19–26, 2003 (cit. on p. 14).

[53] *Apache storm*, `http://storm.apache.org`, 2019 (cit. on p. 14).

[54] P. Carbone, A. Katsifodimos, S. Ewen *et al.*, '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 p. 14).

[55] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez *et al.*, 'Streamcloud: An elastic and scalable data streaming system,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012 (cit. on p. 14).

[56] V. Cardellini, F. Lo Presti, M. Nardelli and G. Russo Russo, 'Run-time adaptation of data stream processing systems: The state of the art,' *ACM Computing Surveys*, 2022 (cit. on p. 15).

[57] V. Gulisano, D. Palyvos-Giannas, B. Havers and M. Papatriantafilou, 'The role of event-time order in data streaming analysis,' in *ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20, Association for Computing Machinery, 2020, 214–217 (cit. on pp. 15, 16).

[58] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou and P. Tsigas, 'Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join,' *IEEE Transactions on Big Data*, vol. 7, no. 2, pp. 299–312, 2016 (cit. on pp. 16–18, 23, 29).

[59] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos *et al.*, 'Viper: A module for communication-layer determinism and scaling in low-latency stream processing,' *Future Generation Computer Systems*, vol. 88, pp. 297–308, 2018 (cit. on pp. 16, 23).

[60] L Lamport, 'Proving the Correctness of Multiprocess Programs,' *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977 (cit. on p. 16).

[61] M. P. Herlihy and J. M. Wing, 'Linearizability: A correctness condition for concurrent objects,' vol. 12, no. 3, 463–492, 1990 (cit. on p. 16).

[62] L Lamport, 'How to make a correct multiprocess program execute correctly on a multiprocessor,' *IEEE Transactions on Computers*, vol. 46, no. 7, pp. 779–782, 1997 (cit. on p. 16).

[63] M. Herlihy, 'Wait-free synchronization,' *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, 124–149, 1991 (cit. on p. 17).

[64] D. Cederman, A. Gidenstam, P. Ha *et al.*, 'Lock-free concurrent data structures,' *Programming Multicore and Many-core Computing Systems*, vol. 86, p. 59, 2017 (cit. on p. 17).

[65] D. Cederman, B. Chatterjee, N. Nguyen *et al.*, 'A study of the behavior of synchronization methods in commonly used languages and systems,' in *IEEE International Symposium on Parallel and Distributed Processing*, 2013, pp. 1309–1320 (cit. on p. 17).

[66] D. Ediger, R. McColl, J. Riedy and D. A. Bader, 'Stinger: High performance data structure for streaming graphs,' in *IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5 (cit. on p. 17).

[67] D. Cederman, V. Gulisano, Y. Nikolakopoulos *et al.*, 'Concurrent data structures for efficient streaming aggregation,' in *ACM symposium on Parallelism in algorithms and architectures*, Association for Computing Machinery, 2014, pp. 76–78 (cit. on p. 17).

[68] V. Gulisano, Y. Nikolakopoulos, D. Cederman *et al.*, 'Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types,' *ACM Transactions Parallel Computing*, vol. 4, no. 2, 2017 (cit. on p. 17).

[69] R. T. H. Collis, 'LiDAR,' *Appl. Opt*, vol. 9, no. 8, pp. 1782–1788, 1970 (cit. on p. 19).

[70] P. Dong and Q. Chen, *LiDAR remote sensing and applications*. CRC Press, 2017 (cit. on p. 20).

[71] S. Gargoum and K. El-Basyouny, 'Automated extraction of road features using LiDAR data: A review of LiDAR applications in transportation,' in *The International Conference on Transportation Information and Safety*, 2017, pp. 563–574 (cit. on p. 20).

[72] J. E. Means, S. A. Acker, B. J. Fitt *et al.*, 'Predicting forest stand characteristics with airborne scanning LiDAR,' *Photogrammetric Engineering and Remote Sensing*, vol. 66, no. 11, pp. 1367–1372, 2000 (cit. on p. 20).

[73] R. B. Rusu and S. Cousins, '3D is here: Point cloud library (pcl),' in *IEEE International Conference on Robotics and Automation*, 2011, pp. 1–4 (cit. on p. 20).

[74] K. Klasing, D. Wollherr and M. Buss, 'Realtime segmentation of range data using continuous nearest neighbors,' in *IEEE International Conference on Robotics and Automation*, 2009, pp. 2431–2436 (cit. on p. 21).

[75] D. Zermas, I. Izzat and N. Papanikolopoulos, 'Fast segmentation of 3D point clouds: A paradigm on LiDAR data for autonomous vehicle applications,' in *IEEE International Conference on Robotics and Automation*, 2017, pp. 5067–5073 (cit. on p. 21).

[76] H. Yin, X. Yang and C. He, 'Spherical coordinates based methods of ground extraction and objects segmentation using 3-D LiDAR sensor,' *IEEE Intelligent Transportation Systems Magazine*, vol. 8, no. 1, pp. 61–68, 2016 (cit. on p. 21).

[77] J. Kletti, *Manufacturing Execution System - MES*. Springer, 2007 (cit. on p. 21).

[78] M. Subramaniyan, A. Skoogh, M. Gopalakrishnan *et al.*, 'An algorithm for data-driven shifting bottleneck detection,' *Cogent Engineering*, vol. 3, no. 1, p. 1 239 516, 2016 (cit. on p. 21).

[79] L. Li, Q. Chang and J. Ni, 'Data driven bottleneck detection of manufacturing systems,' *International Journal of Production Research*, vol. 47, no. 18, pp. 5019–5036, 2009 (cit. on p. 21).

[80] B. Krzanich, *Data is the new oil in the future of automated driving*, `https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving`, 2016 (cit. on p. 22).

[81] C. Batini, C. Cappiello, C. Francalanci and A. Maurino, 'Methodologies for data quality assessment and improvement,' *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–52, 2009 (cit. on p. 23).

[82] H. Elmeleegy, A. K. Elmagarmid, E. Cecchet *et al.*, '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 p. 23).

[83] Q. Xie, C. Pang, X. Zhou *et al.*, 'Maximum error-bounded Piecewise Linear Representation for online stream approximation,' *The VLDB Journal*, vol. 23, no. 6, pp. 915–937, 2014 (cit. on p. 23).

[84] G. Luo, K. Yi, S.-W. Cheng *et al.*, 'Piecewise linear approximation of streaming time series data with max-error guarantees,' in *IEEE international conference on data engineering*, 2015, pp. 173–184 (cit. on p. 23).

[85] 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 p. 23).

[86] F. Grützmacher, B. Beichler, A. Hein *et al.*, 'Time and memory efficient online Piecewise Linear Approximation of sensor signals,' *Sensors*, vol. 18, no. 6, p. 1672, 2018 (cit. on p. 23).

[87] B. Gedik, R. R. Bordawekar and S. Y. Philip, 'CellJoin: a parallel stream join operator for the cell processor,' *The VLDB Journal*, 2009 (cit. on p. 23).

[88] P. Roy, J. Teubner and R. Gemulla, 'Low-latency handshake join,' *Proceedings of the VLDB Endowment*, 2014 (cit. on p. 23).

[89] C. Hochreiner, M. Vögler, S. Schulte and S. Dustdar, 'Elastic stream processing for the Internet of Things,' in *IEEE International Conference on Cloud Computing*, 2016, pp. 100–107 (cit. on p. 23).

[90] V. Cardellini, M. Nardelli and D. Luzi, 'Elastic stateful stream processing in storm,' in *International Conference on High Performance Computing & Simulation*, 2016, pp. 583–590 (cit. on p. 23).

[91] B. Gedik, 'Partitioning functions for stateful data parallelism in stream processing,' *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014 (cit. on p. 23).

Part II

Thesis Chapters