Hasso Plattner Institute
Chair for Data Engineering Systems

in cooperation with

Technische Universität Berlin
Chair for Database Systems and Information Management

Master Thesis

Efficient Distributed In-Network Window Aggregation

Effiziente verteilte, fensterbasierte Datenstromaggregation im Netzwerk

Lawrence Benson
Matriculation Number: 771109

Supervisor
Prof. Dr. Tilmann Rabl (HPI)

Advisor
Philipp Grulich, M.Sc. (TU)

01.11.2019

Abstract

Over the past decades, the adoption of sensors and the Internet-of-Things (IoT) has increased rapidly. In order to make sense of the high-volume event streams, efficient distributed data stream processing is needed. To analyze continuous data, events are often grouped into timed-based chunks known as *windows*. For stream analysis, modern Stream Processing Engines (SPEs) support complex window types and user-defined aggregation functions. Most SPEs run on a cluster in a central data center. However, many IoT use-cases are highly distributed and SPEs are agnostic of this distribution, requiring central processing. Wireless Sensor Networks, on the other hand, actively leverage the distributed nature of the data streams in their aggregation, but support only basic aggregation and windows.

To bridge the gap between complex central aggregations and simple distributed ones, we propose Disco, a stream processing framework that supports complex in-network window aggregation. We present a model that processes complex window types on multiple independent nodes while efficiently aggregating their data. We implement this model in Disco, which leverages the distributed nature of incoming event streams and the computational power of the nodes they pass through towards the central data center. Our evaluation shows that Disco's throughput scales linearly with the number of nodes and that Disco already outperforms a centralized solution in a two-node setup. Furthermore, Disco reduces the network cost significantly compared to the centralized approach. Disco's tree-like topology handles thousands of nodes per level and scales to support future data-intensive streaming applications.

Zusammenfassung

In den letzten Jahrzehnten hat die Verbreitung von Sensoren und dem Internet der Dinge (IoT) rasant zugenommen. Um die Ereignisströme zu verstehen, ist eine effiziente verteilte Datenstromverarbeitung erforderlich. Für die Analyse von kontinuierlichen Daten werden Ereignisse häufig in zeitgesteuerten Blöcken gruppiert, die als *Fenster* bezeichnet werden. Für die Datenstromanalyse unterstützen moderne Stream Processing Engines (SPEs) komplexe Fenstertypen und benutzerdefinierte Aggregationsfunktionen. Die meisten SPEs laufen auf einem Cluster in einem zentralen Rechenzentrum. Viele IoT-Anwendungsfälle sind jedoch stark verteilt, was von SPEs nicht berücksichtigt wird und dadurch eine zentrale Verarbeitung erfordert. Drahtlose Sensornetzwerke hingegen nutzen aktiv den verteilten Charakter der Datenströme in ihrer Aggregation, unterstützen aber nur einfache Aggregationsfunktionen und Fenster.

Um die Lücke zwischen komplexen zentralen und einfachen verteilten Aggregationen zu schließen, präsentieren wir Disco, ein Framework für Datenstromverarbeitung, das die komplexe Aggregation von Fenstern im Netzwerk unterstützt. Wir präsentieren ein Modell, das komplexe Fenstertypen auf mehreren unabhängigen Knoten verarbeitet und gleichzeitig deren Daten effizient aggregiert. Wir implementieren dieses Modell in Disco, das die Verteilung eingehender Ereignisströme und die Rechenleistung der Knoten, die sie zum zentralen Rechenzentrum durchlaufen, nutzt. Unsere Auswertung zeigt, dass der Durchsatz von Disco linear mit der Anzahl der Knoten skaliert und dass Disco eine zentralisierte Lösung bereits mit Zweiknoten-Setup übertrifft. Darüber hinaus reduziert Disco die Netzwerkkosten im Vergleich zum zentralisierten Ansatz erheblich. Die baumartige Topologie von Disco bewältigt Tausende von Knoten pro Ebene und kann skaliert werden, um zukünftige datenintensive Datenstromanwendungen zu unterstützen.

Contents

1 Introduction **1**

1.1 Motivational Example 4

1.2 Contribution 6

1.3 Scope and Delimitation 6

1.4 Thesis Outline 7

2 Background **8**

2.1 Fog Computing 8

2.2 Wireless Sensor Networks 9

2.2.1 WSN Structure 10

2.2.2 WSNs in Fog Computing 12

2.3 Stream Processing Engines 13

2.3.1 SPE Core Concepts 13

2.3.2 SPEs in Fog Computing 17

2.4 Windowing in Data Streams 17

2.4.1 Window Measures 18

2.4.2 Window Types 19

2.5 Window Aggregation 21

2.5.1 Aggregate Functions 22

2.5.2 Data Aggregation in Wireless Sensor Networks 24

2.5.3 Aggregate-Sharing in Stream Processing Engines 26

2.6 General Stream Slicing 29

2.7 Summary 32

3 Distributed Data Aggregation Concepts **33**

3.1 Definitions 33

3.2 Window Types 35

3.3 Aggregate Functions 43

4 Distributed Data Aggregation Implementation **46**

4.1 Disco Architecture Overview 46

4.2 Window Representation 49

4.3 Window Types 51

4.4 Aggregate Functions 55

4.5 Disco Components 57

4.6 Summary 60

5 Evaluation **62**

5.1 Setup and Methodology 62

5.2 System Benchmarks 64

5.2.1 Scalability 64

5.2.2 Network Cost 67

5.2.3 Concurrent Windows 68

5.2.4 Concurrent Keys 70

5.2.5 Window Latency 71

5.2.6 Child Node Performance 72

5.2.7 Root Node Performance 73

5.3 Discussion 74

6 Related Work **76**

6.1 Aggregate-Sharing in Stream Processing Engines 76

6.2 Distributed Aggregation in WSNs 78

7 Conclusion **82**

1 Introduction

The adoption of sensors and IoT devices has increased rapidly over the past decades, leading to a high demand for efficient distributed data stream processing. By 2020, more than 20 billion *Things* will be connected to the internet [30] and many applications consume this data already. Use-cases range from smart health applications [5] to connected driving [63] and entire smart cities [7]. However, all of these applications deal with the common problem of how to process distributed, high-velocity data in a cost-efficient manner.

Many areas of research are currently focusing on how to efficiently overcome these challenges. One of these areas is *Fog Computing*, which deals with the computation of data at the edge of the network, along its path to the cloud, and eventually in the cloud. IoT sensors produce data constantly at high rates, making it infeasible to send all data to a central data center, which is often not located geographically close to the edge devices. The aim of Fog computing is to bridge the gap between processing data only at the edge, without global knowledge of the system and processing data only in a central cloud, where large amounts of data need to be transferred across the globe. Some of the main advantages of Fog computing compared to only Cloud computing are a reduced latency and lower network costs due to less traffic being sent to a central cluster [66]. However, this also leads to combined problems from both Edge and Cloud computing, such as distributed data, resource provisioning, and centralization (typical in Cloud computing) with heterogeneous compute resources, low bandwidth, and low availability (typical in Edge computing) [53, 68].

As data is produced continuously at multiple sensors in IoT networks, we can naturally think of their analysis as a distributed stream processing application. Lots of data is continuously produced at all kinds of sensors, making it impossible for humans to understand it without powerful analysis tools and frameworks due to its sheer volume and velocity. A large group of frameworks that allow for efficient data processing are Stream Processing Engines (SPE). SPEs are well equipped for analyzing volatile, high-velocity IoT data due to efficient data aggregation modules. Aggregation can be performed on millions of events per hour on modern SPEs [49, 54]. However, even though the most notable SPEs such as Apache Flink [8], Apache Spark Streaming [70], or Apache Storm [57] are all distributed engines, they all run within a single centralized data center. Thus, requiring all raw data to be sent through the network to a compute-cluster in which the data is aggregated. As these clusters are located in a single data center, e.g. Amazon US-East or on-premise, and the sensors producing data are usually

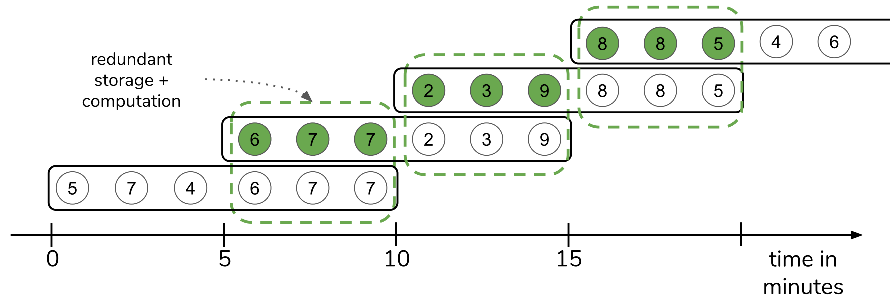


Figure 1: Redundant overlap in windows.

distributed throughout a larger region, e.g. an entire continent or even worldwide, the cost of transporting raw data points becomes a limiting factor [62]. As this directly contradicts the goal of moving computation closer to the edge, distributed aggregation poses a challenge for existing SPEs. We can solve this challenge by supporting a distributed, close-to-edge aggregation model. In this work, we aim to adapt common aggregation techniques to run efficiently in a Fog-like network structure.

To efficiently aggregate unbounded data streams, SPEs use *windows* to define bounded subsets of the stream. A window is a finite, logical view of the stream that is usually characterized by time, e.g. the last ten seconds, five minutes, or two hours, on which aggregation functions are then applied. Also, it is not always possible to correctly aggregate unbounded data, e.g., calculate the 95th-percentile, making windows a required concept for non-trivial data aggregation. Thus, most modern SPEs use windows to allow for complex analysis tasks. Such a window is generally defined by the user of an SPE within a query, e.g. calculate the average stock price for the last ten minutes. However, complex data analysis frequently requires more than one query to provide the user with the expected result. Large applications may have hundreds of windows to constantly aggregate their data. However, many queries often refer to the same data stream, thus reading and processing the same raw data multiple times.

A simple example query, such as "Calculate the average stock price of the last 10 minutes, every 5 minutes" already shows the redundant data processing problem within a single window, as shown in Figure 1. In this query, there is a five-minute overlap between each adjacent window. Here, the circles represent the current price of a stock as an event. The naive approach is to store all raw stream events for each window and compute the aggregate once the window is complete, i.e. every five minutes. If we calculate the average price for the window from minute 0 to minute 10 (bottom left), we come up with $(5+7+4+6+7+7)\ /\ 6 = 6$. If we do the same

for the next window, we look at $6 + 7 + 7$ again. The naive approach leads to a high computational and storage redundancy, as the tuples for the overlapping five minutes are stored and computed twice. If we add additional overlapping queries to this system, the cost of this approach quickly becomes intolerably high.

Recent research in stream processing presents multiple ways to process overlapping windows more efficiently by applying aggregate-sharing techniques [9, 40, 58]. Yet, these approaches deal with window aggregation as a central task to be performed on a single node in a single data center. We propose a distributed approach in which we leverage existing research and adapt it to push down the computational tasks closer to the edge. For this, we discuss these single-node methods in more detail in Section 2.5.3 and show how they can be used in a distributed setting.

The main focus of Edge computing is to not send all data to a central processing instance. Thus, streaming windows need to be created and processed along the network path from the data source to the querying node. Previous research in the area of Wireless Sensor Networks (WSN) describes methods of how to enable simple window aggregations in networks with limited bandwidth and processing capacity [45, 65]. However, current approaches focus only on very simple windowing techniques to reduce bandwidth. To support more advanced windows, as required by modern stream processing workloads, these approaches need to be extended. Especially taking into consideration that the network does not consist of homogeneous sensors but of heterogeneous hardware with increasing capacity towards the cloud [27]. This allows for more complex processing along the path than is possible in traditional WSNs.

Currently, most real-world streaming applications work on one or a few large data streams, e.g. a stream of all stock prices (as seen in Figure 1) or a click-stream of user events on a website. In the context of Edge and Fog computing, however, the data is provided by a large number of independent sensors or Internet of Things (IoT) devices. Compared to the central aggregation approach of modern SPEs, we aim at processing the data closer to the edge and, thus, all streams cannot be merged into a single big one. For this, we must adapt existing techniques to handle the high number of streams available and find more efficient ways to process them. This entails a new group of problems compared to the single-stream SPEs; namely, aggregations are not performed on windows of a single stream anymore but on windows of multiple streams. This raises three central questions, which we discuss and answer in Sections 3 and 4.

1. How are windows defined on multiple streams?
2. How can each stream be independently processed closer to the edge?
3. How can we efficiently aggregate and merge windows of distributed streams?

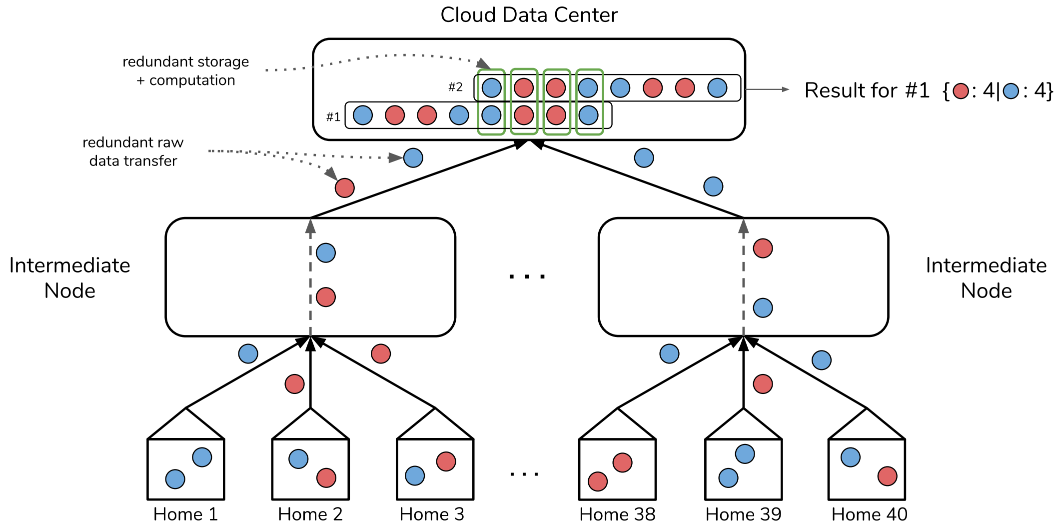


Figure 2: Centralized smart home streaming application.

1.1 Motivational Example

To illustrate a Fog processing application, we use the dataset of the DEBS Grand Challenge 2014 [34], which deals with smart homes. In this system, there are 40 houses containing a total of 2125 smart plugs. The data collected over the course of a month totals over four billion tuples. Each house has an average of 53 smart plugs and each plug generates just under one tuple per second. This is a rather small example, as most IoT devices will be present in more than 40 houses. In a setting with 400,000 homes, the approximate 1500 events per second here quickly multiply to 15 million events per second. Taking the distributed locations of the houses into account, it quickly becomes infeasible to collect all data at a central compute-cluster. Thus, a decentralized streaming approach to analyze the sensor data should be used.

Figure 2 shows this smart home application as it is currently done by most modern SPEs. All data is passed through the network hops to the central compute-cluster in the cloud. Note, that in this approach all computational power of the intermediate nodes is wasted as they simply act as tuple forwarders. Also, the load on the compute-cluster in the data center is pretty high, as all raw events are collected and processed there. As described above, the naive approach stores and computes each tuple for every window it is in, which can be seen in the green squares in the data center of Figure 2.

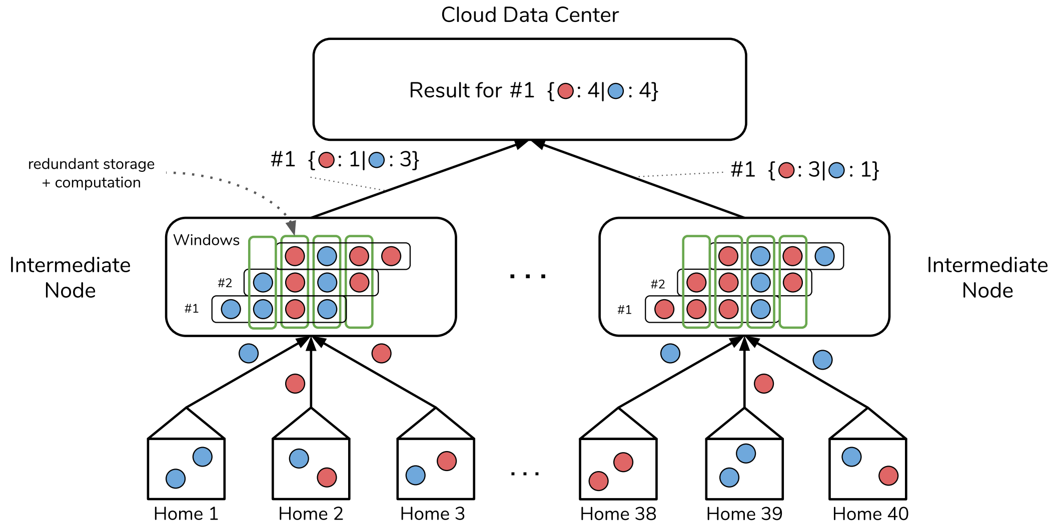


Figure 3: Distributed smart home streaming application.

A decentralized application is shown in Figure 3. Compared to the central approach above, the raw data is processed by the intermediate network hops. This reduces the network cost drastically, as each tuple is sent only to the intermediate node that processes it. The node then only passes on aggregated values to the cloud. There, the application only has to merge the pre-aggregated values from the previous network hops and thus, reducing its load heavily. As in Figure 2, there is redundancy in storage and computation for each tuple processed. In Section 4, we discuss how to avoid this redundancy if possible. The main advantages of the distributed approach, compared to the central one, are the reduced network consumption and the reduced load on the compute-cluster. We evaluate the differences between the two approaches in detail in Section 5.

1.2 Contribution

To overcome the high network cost of current centralized data aggregation approaches, we propose Disco, a stream processing framework that supports complex in-network window aggregation. With Disco, we leverage existing aggregate-sharing techniques to reduce computational and storage redundancy on single nodes while also reducing the amount of data that needs to be transferred between nodes for correct aggregation results.

In summary, we contribute the following:

1. We formalize streaming window types on multiple (distributed) streams.
2. We propose Disco, a new windowing aggregation framework that allows for efficient decentralized in-network data aggregation.
3. We evaluate Disco extensively, showing that in many cases a decentralized data aggregation approach outperforms a centralized one in both network cost and throughput.

1.3 Scope and Delimitation

The main focus of this thesis is to explore distributed data aggregation for arbitrary window types. For this, we built the prototypical distributed Stream Processing Engine Disco. We note here that Disco is not comparable to modern SPEs in many regards, as we make simplifying assumptions to reduce the complexity of the prototype. For example, Disco only supports numerical data types, as the main use-case for data aggregation is numeric. Also, we did not include custom queries but only simple window- and aggregation-interfaces. Unlike most modern SPEs, we do not handle network or node failure, nor do we checkpoint intermediate results and states. The integration of all this poses interesting areas of future work, especially dealing with fault-tolerance in a tree-like network structure. For the sake of simplicity, we use a custom string-based protocol to send data between the nodes. A more specialized binary protocol would further reduce the network cost but comes at the cost of less flexibility during the implementation.

1.4 Thesis Outline

Section 2
In Section 2, we introduce the technical concepts on which our contribution is based. We start by introducing Fog computing, Wireless Sensor Networks, and Stream Processing Engines. Then, we present data and window aggregation concepts in stream processing, followed by a brief explanation of *general steam slicing* [58], as it is a foundation of Disco.

Section 3
In Section 3, we present the concepts behind distributed window aggregation on multiple streams. For this purpose, we first introduce some window aggregation definitions, followed by a detailed discussion on distributed merging semantics for different window types and aggregation functions.

Section 4
In Section 4, we present our prototype Disco based on the concepts introduced in the previous section. We fist present an architectural overview of Disco. We then discuss the implementation for different window types and aggregations, followed by detailed explanations of Disco's components.

Section 5
We extensively evaluate Disco in Section 5. For this purpose, we evaluate the latency and throughput performance of Disco compared to a centralized data aggregation approach for various window and aggregation types. We also evaluate the total network cost of centralized and decentralized solutions, as well as micro-benchmarks for the performance of single components.

Section 6
In Section 6, we present an overview of related research in the area of streamed data aggregation. For this purpose, we discuss aggregation in modern SPEs, followed by research on aggregate-sharing in SPEs and distributed aggregation in WSNs.

Section 7
In Section 7, we summarize our work and give an outlook towards future work in the area of distributed data and window aggregation.

2 Background

In this section, we cover the technical concepts upon which our contribution builds. First, we discuss Fog computing (see Section 2.1) as the underlying network structure for our distributed aggregation. Then, we cover Wireless Sensor Networks in Section 2.2, as they build the basis for Fog networks consisting of many edge devices. In Section 2.3, we present Stream Processing Engines as modern systems for complex data analytics on continuous data streams. To analyze finite chunks of unbounded data streams, Stream Processing Engines apply windowing techniques, which we discuss in Section 2.4. We then present windowed data aggregation techniques in Section 2.5 as the basis for our distributed aggregation. Finally, we present a general stream slicing approach (see Section 2.6) that allows us to efficiently create arbitrary windows on single nodes, followed by a summary in Section 2.7.

2.1 Fog Computing

With the rise of the Internet of Things (IoT) over the last few years and its predicted growth of anywhere between 3.5 and 20 billion connected devices in the next few years [10, 30], new network structures are emerging to efficiently handle this IoT data. One new central concept is Fog computing, which combines edge and cloud computing. While edge computing deals with the management of individual data directly at the IoT devices [21], cloud computing deals with a more centralized management of all data in one or a few a cloud data centers [61]. To bridge the gap between these two approaches, Fog computing deals with management, storage, and processing of data on network nodes between the IoT devices and the data center [68]. The main advantage of such Fog networks is the improved latency and network cost due to the close physical proximity of Fog servers and IoT sensors. However, these improvements are not always trivially achieved, as they require a good understanding of the placement and locations of the network components [12, 48].

Figure 4 shows the structure of such a Fog network. Here, the IoT devices or sensors communicate with applications that are located close to them. This could be, e.g. a control panel for a smart home or a digital health application. The applications then communicate with the intermediate Fog servers where data can be stored and analyzed. The Fog server level can also span multiple levels. Finally, the Fog servers communicate with the central cloud data center where only very little raw data arrives and most of the processed data is already aggregated in some

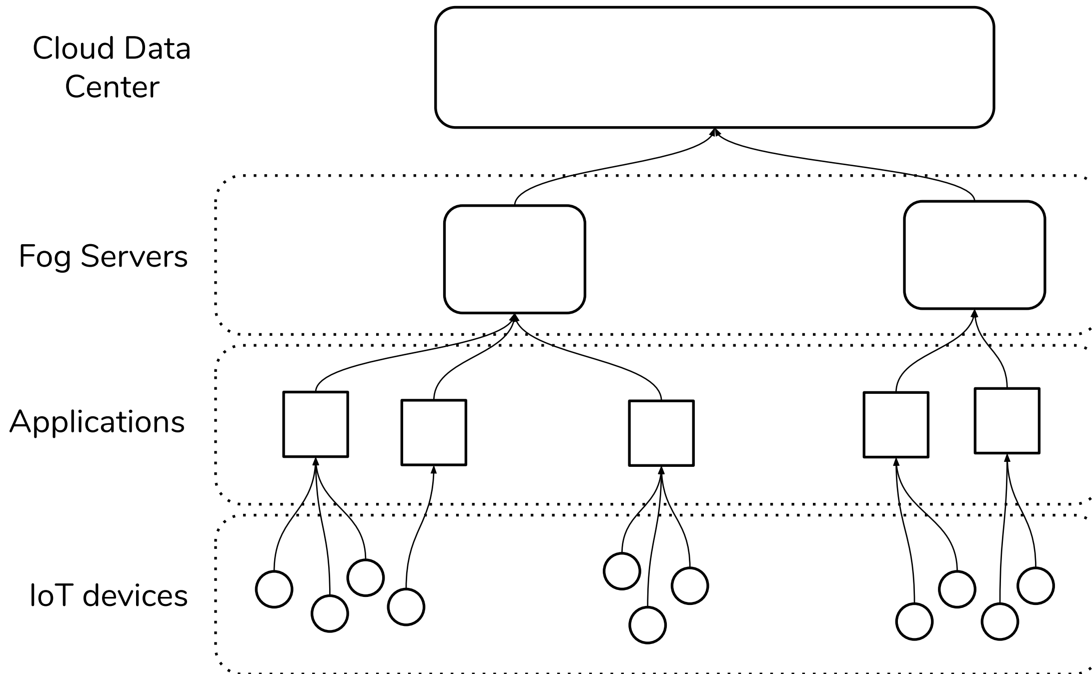


Figure 4: Fog network structure.

form. For example, Gia et al. [23] propose a smart heath system in a Fog structure in which ECG data is transmitted to a nearby Fog gateway and analyzed there. Only the results are then transmitted further to be accessible for other applications from the cloud. They achieve a 93% reduction in data transferred to the cloud while providing a low latency analytical result.

For this work, we make use of a Fog computing network structure. Many small sensor devices build the ground level of our network topology, followed by intermediate network servers on the way to a global cloud data center. We leverage this multi-level structure to avoid sending all of the raw data to a central location, which allows us to provide a low latency with a reduced network cost. We discuss the use of Fog structures in our proposed solution in Section 4.

2.2 Wireless Sensor Networks

To better understand the lower levels of Fog computing, we take a step back and discuss Wireless Sensor Networks (WSN). Wireless Sensor Networks consist of many small electronic sensing devices or sensors. These sensors usually consist of three main components: sensing, processing, and communicating [2]. A WSN can

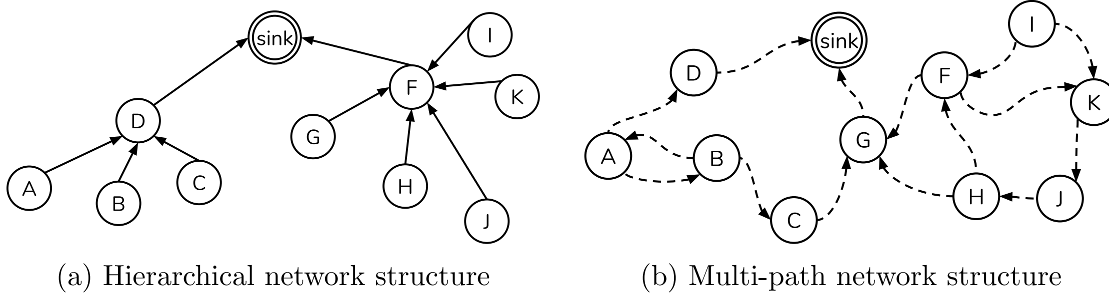
easily contain hundreds or thousands of such sensors that autonomously create a network and communicate with each other. The use cases of Wireless Sensor Networks range from military surveillance [2] over medical diagnosis assistance [67] to smart homes [52], making them widely applicable. One main commonality in all ranges of applications is the continuous production of data. Depending on the rate of the sensors, a single device can send between one and hundreds of events per second [45]. In networks with hundreds or even thousands of sensors, this rapidly adds up to tens or hundreds of thousands of messages per second broadcast through the WSN.

2.2.1 WSN Structure

The sensors in WSNs are small devices with very little processing power and limited memory capacity. The communication is often performed via low-energy channels, such as radio [64] or mobile networks [29]. These communication channels are very limited in bandwidth and are run in lower power mode to preserve the battery of the sensor devices. Taking the three main hardware limitations (low CPU, low memory, low bandwidth) into account, it is infeasible to communicate with all other sensors. Thus, chains of communicating nodes or node-groups are created. The emerging network structures are discussed below and can be seen in Figure 5. No sensor will see all of the data, as each node usually communicates with only a few other nodes. This leads to a strongly distributed nature of the network. Nodes may fail at any given time thus cutting connections randomly. Wireless Sensor Networks need to account for such common failures by being integrated into systems that do not rely on the availability of any single sensor or node. These systems are often heavily redundant to account for random loss of devices.

Akyildiz et al. [2] show many use-cases that rely on redundancy to perform correct WSN calculations in a survey. One such application is by Liu et al. [44] who use WSNs to predict volcanic activity by using many inexpensive sensors to overcome the lack of signals from other nodes in close proximity. Their prediction requires a large number of measurements and not all nodes can continuously provide their coordinator nodes with data, so redundancy is used to ensure a steady flow of data.

To interact with WSNs from outside of the system, the results of the application need to be accessible via certain nodes (so-called *sinks* or *base stations*). One main challenge in Wireless Sensor Networks is to efficiently create routing structures between the sensor nodes to reach a sink. The routing structures are usually self-organized by the nodes. In the volcanic activity application mentioned above,



(a) Hierarchical network structure (b) Multi-path network structure

Figure 5: Main network structures in Wireless Sensor Networks.

the user interacts with coordinator nodes that perform the activity prediction on received raw data. These coordinator nodes are sinks. Depending on the application, this structure can be fixed, i.e. nodes always send their data to the same node, or dynamic, i.e. the structure changes constantly and thus the sensors send their data to varying nodes. However, *fixed* does not necessarily imply that the structure never changes, it is just assumed to change less frequently than *dynamic* structures. If nodes are positioned stationary, the network will likely not change as often as a network of sensors attached to vehicles that drive around.

In Figure 5 we see the two main network structures of WSNs. Figure 5a shows a hierarchical or tree-like structure. In this, sensors communicate with parent nodes along the path to the sink. Nodes A, B, and C send their data to parent node D. D then again sends the collected data or a computational result to its respective parent, in this case, the sink node. All data of that sub-tree can then be accessed via the sink. This applies analogously to nodes F through K. There are many protocols that achieve a network structure like the one in Figure 5a. A lot of research focuses on how to achieve this structure efficiently with maximized bandwidth. We refer to previous work [26, 32, 64], which all deals with finding hierarchical routing algorithms. As an intuition, a trivial hierarchical algorithm is

1. Node n joins network
2. n broadcasts to all nodes in close proximity
3. n receives answer from one or more other nodes $o_1, o_2, ..., o_k$
4. n selects node, from which the first answer arrived, as its parent p_n
5. n sends all data to parent p_n from now on

The main advantage of a hierarchical approach is efficient bandwidth use. Each node can send data along the best path that was determined by a routing algorithm. However, should a parent node fail, the affected nodes need to re-run the

routing algorithm to determine a new parent. Depending on the algorithm, this can be an expensive re-computation.

In contrast to a hierarchical structure, *gossip-based* or *multi-path* routing algorithms construct networks as can be seen in Figure 5b. In these multi-path structures, each node sends its data to one or more nodes in close proximity, possibly through a broadcast. In the above figure, node B sends its data to nodes A and C. They, in turn, send their data or a computational result to nodes D and G, respectively. We note here that loops are possible in such a network, as can be seen between nodes A and B, as well as in the cycle $K \rightarrow J \rightarrow H \rightarrow F \rightarrow K$. The system needs to handle such duplicates correctly, in order to avoid network flooding. Eventually, the data is propagated to the sink node, where it can be accessed from outside of the system.

The main advantage of a multi-path structure is its natural redundancy. Should a single node fail, the data is still transmitted to other nodes and thus propagates to the sink. This, however, comes at the cost of duplicate network transmission, which can lead to a high network contention and thus reduce the speed of the network. We refer to previous work [46, 50], which deals with different multi-path routing algorithms, their advantages, and their disadvantages in detail.

2.2.2 WSNs in Fog Computing

Even though the research of Wireless Sensor Networks goes back two decades, it is still applicable today. With the rise of the Internet of Things (IoT) and Fog computing, sensors are used widely across the globe. The use-cases of WSNs expand to everyday life, as IoT devices are located in homes, in cars, or on humans. One main difference between WSNs and Fog computing, however, is the geo-distribution. Sensors in a WSN are often located in close proximity to detect certain behavior specific to that geographical region. The *Fog*, on the other hand, can spread across entire cities or countries, with applications such as smart cities. Another difference, which follows directly from the geo-distribution, is the sink location. The sink in WSNs is usually part of the WSN at a given location. An IoT sink, contrarily, is often located in the cloud, to allow for a global view of all devices of a certain application. Thus, Fog networks usually follow a hierarchical routing structure from the (smart-)device to the cloud. Our motivational example in Section 1.1 can be interpreted as a WSN in some ways. Each smart-home sensor communicates with other sensors in a hierarchical fashion and the data is finally collected at a sink in the cloud.

In this thesis, we focus on the hierarchical, tree-like structures of Wireless Sensor Networks. We also work on an already constructed network, as our solution is agnostic of the pathfinding algorithm and does not have any impact on it. The main advantage of multi-path structures in WSNs, fault-tolerance, is not within the scope of this thesis and is left for future work.

A lot of research focuses on how to work efficiently within the three main hardware limitations – low computational power, low memory capacity, and low bandwidth – by reducing either the required bandwidth [45, 65] or the necessary computation [44]. One of these research areas focuses on data aggregation, which we discuss in detail in Section 2.5.2. The goal of this research is to reduce the volume of raw data that is sent through the network and thus reduce the cost of computation on the sink node of the network.

2.3 Stream Processing Engines

A Stream Processing Engine (SPE) is an application that performs computational tasks on continuous data streams. With the steady increase in data velocity over the past decades, SPEs are becoming more and more popular. Companies want to analyze their data as soon as it arrives, to make near-real-time business decisions. Typical use-cases for such instant data processing are in social networks, fraud detection, or marketing [4, 51]. Stream Processing Engines can work on large amounts of data, up to petabyte scale [54]. To handle these data volumes, SPEs are often deployed in cloud environments with either one very large server (scale-up) or many commodity servers (scale-out).

In the remainder of this Section, we present the core concepts of SPEs (see Section 2.3.1) and how SPEs integrate with Fog computing (see Section 2.3.2).

2.3.1 SPE Core Concepts

At this point, we present some core concepts of SPEs to provide insight into how our work integrates with them. For this, we first discuss scale-up vs. scale-out and stream- vs. batch-processing, followed by stream queries and stream operators.

Scale-Up vs. Scale-Out
Scale-up describes the concept of using a single server with a very high memory capacity and high processing power. The aim of scale-up systems is to avoid communication and synchronization overhead, which is required in distributed systems. Two common scale-up systems are Saber [37] and StreamBox [47]. Scale-out, on the other hand, uses many commodity servers to each solve small parts of the problem. One main advantage of scale-out is its elasticity, the number of servers can be increased or decreased rapidly to account for a higher or lower volume of data. The hardware used for these scale-out servers is usually homogeneous with a high bandwidth, to allow for fast communication. Depending on the exact task, data can be either processed completely distributed or needs to be collected and merged at some point. A map-only task that writes directly to a database, for example, does not need all data to be in one place, whereas an aggregation requires central data collection. This central data collection is similar to the *sinks* in Wireless Sensor Networks (see Section 2.2.1). However, having a single sink is not always required as individual nodes can also act as sinks and emit data from the system. Common scale-out systems are Apache Spark Streaming [70], Apache Flink [8], and Apache Storm [57]. We refer to previous work [31, 71], which deals with the different advantages and disadvantages of scale-up and scale-out systems. In this work, we focus on scale-out systems as we leverage the distributed nature of such systems.

Stream-Processing vs. Micro-Batch-Processing
Stream Processing Engines work on continuous data streams. However, there are two main approaches to achieve this – event-at-a-time and micro-batching. Event-at-a-time processing describes a system in which each event is processed individually. As soon as an event enters the SPE, it is processed and passed through the application. We refer to this approach as *stream-processing*. On the other hand, for micro-batching, the SPE collects entering events for a certain period and passes them through in a batch.

In Figure 6 we see both stream- and micro-batch-processing. Figure 6a schematically shows how data that flows into the SPE is passed on event-by-event. The SPE buffers a few events and passes them to the operators when they are ready. Each operator only sees one event at any given time. In contrast, Figure 6b shows the same flow but with micro-batches. The SPE collects events in its buffer and groups them into batches, in this example of size three. Each operator then sees these batches instead of single events. We note here that these are just the rough concepts behind stream- and micro-batch-processing, as many aspects are heavily implementation-dependent. However, the intuition of both methods should be understood.

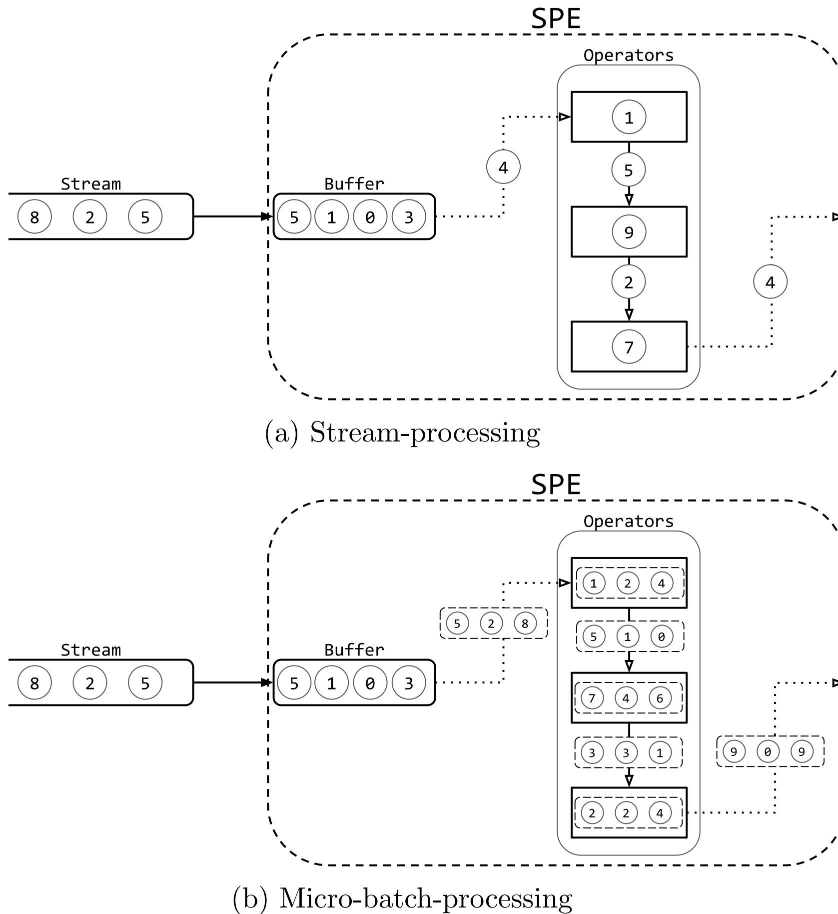
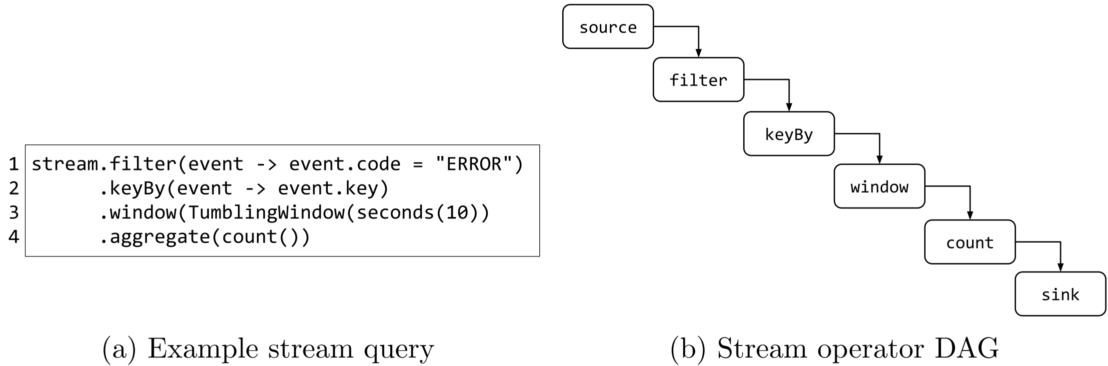


Figure 6: Stream- and micro-batch-processing in SPEs.

The main trade-off between the two approaches is latency vs. throughput. Stream-processing usually accounts for a lower latency, as each tuple is processed as quickly as possible. However, this may come at the cost of reduced throughput, as processing single events incurs a higher overhead in communication and typically does not utilize the given hardware as efficiently as batches do. Micro-batching produces the reversed effect, where latency may increase due to events waiting in the buffer before a batch is created. The throughput may be higher though, because of lower communication overhead and better hardware utilization. We refer to previous work that deals extensively with both approaches and their trade-offs [8, 36, 69, 70, 71].

We note that the two most widely used SPEs support both stream-processing and micro-batching to give the user a choice in the above-mentioned trade-off between



(a) Example stream query (b) Stream operator DAG

Figure 7: Example query and resulting DAG.

latency and throughput. The first versions of Apache Spark supported only micro-batch-processing [69] and later added a Spark Streaming component [70]. Apache Flink supports both approaches but focuses primarily on stream-processing [8].

In this work, we focus on stream-processing as we aim to provide low latency throughput.

Stream Queries
To understand how data is processed by a Stream Processing Engine, we discuss queries on streamed data. A stream query is a directed acyclic graph (DAG, also called dataflow graph) of operators. In this DAG, vertices are the operators and edges are data streams. Operators (vertex) in a DAG continuously process incoming data and streams (edges) represent unbounded sequences of events flowing between these operators. These streams may be infinite and the DAG can theoretically run forever.

Stream Operators
The basis for any stream query are stream operators. These operators continuously receive event data, process it, and emit a computational result. There are many types of standard operators and an operator can be an arbitrary piece of code that fulfills the criteria of an operator. Thus, SPEs can perform complex computations and aggregations on the data that it receives.

Figure 7 shows an example query and its resulting operator DAG. The query in Figure 7a counts the number of error-events for every key for the past 10 seconds. It does this by calling `filter` on the stream to remove all events that are not errors (Line 1). Then it determines the key for each event (Line 2) so that it can be used for aggregation later. It then creates a window of 10 seconds (Lines 3) in which the aggregation shall take place. Finally, it counts the number of error-events per

key (Line 4) in the given window. We discuss windows and aggregation in detail in Sections 2.4 and 2.5, respectively. This query is translated to the operators in Figure 7b. The *source* and *sink* operators are special operators, that simply ingest and emit events. The operators then perform the above-mentioned task continuously for each event.

In a scale-out SPE with multiple servers, the operators are distributed throughout the system to allow for better load balancing. We make use of this concept in our work, by placing windowing and aggregation operators closer to the raw event stream. We discuss this in detail in Section 4. The conversion from a query to its corresponding operators is not within the scope of this work. We refer to previous work that deals with this task in detail [6, 28, 33].

2.3.2 SPEs in Fog Computing

Stream Processing Engines are an essential part of Fog computing. They represent the last step on the path from the edge devices (e.g. smart sensors) to the cloud. SPEs can process all data that the devices produce and provide the application user with the requested computational results. In contrast to Wireless Sensor Networks, SPEs are usually not limited by low memory or CPU power. Thus, making the combination of WSNs and SPEs ideal for Fog computing. The devices self-organize at a low level, efficiently communicating throughout the network and then the SPE can perform computationally expensive tasks on all data. Again, our motivational example from Section 1.1 can be interpreted as such a Fog computing network. On top of the WSN components, discussed in Section 2.2.2, we can perform arbitrarily complex queries on the data. In this case, the WSN sink is an SPE in the cloud, allowing for a global view across all devices.

2.4 Windowing in Data Streams

Both Wireless Sensor Networks and Stream Processing Engines deal with continuous data streams. These data streams contain a possibly infinite number of events. Analyzing an unbounded stream of events is not always possible, as certain analytical tasks need to see all data to produce correct results. Thus, in both WSNs and SPEs the concept of *windowing* is used. A window simply describes a bounded subset of events on which analytical tasks are possible. Certain operations do not require a bounded subset, e.g. a filter that simply looks at an event and passes it on based on a specified criterion. An aggregation, on the other hand, is impossible

on an infinite set (e.g. median event value). In this work, we focus on aggregation in data streams, so all of our operations require windowing. We formally describe windows in Section 2.4.2, but first, we introduce the concept of time in streams (see Section 2.4.1).

2.4.1 Window Measures

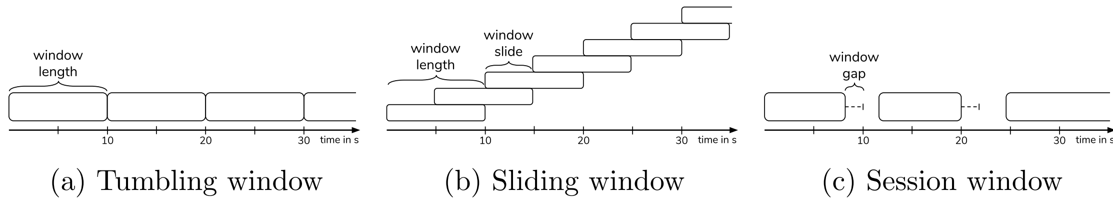
To create a bounded subset of an infinite sequence of events, bounds need to be defined. These bounds determine the window measure, of which the most commonly used one in stream processing is time. For example, a query could analyze data from the last ten minutes of events, the last hour, or the last ten seconds every five seconds. However, time in stream processing is not always clearly defined. 'The last ten minutes' can refer to the time at which the event was created or the time at which the system processes the event. We distinguish between *event-time* and *processing-time* to describe these two concepts [1, 8, 58].

- **Event-time** describes the time at which an event was generated at its source. This can be, e.g. the wall time of a sensor or a logical time added by an application. It does not necessarily need to represent real time and it does not change during processing.
- **Processing-time** describes the time at which an event is processed in the systems. It is usually set as the system time of the application that is currently handling it. Thus, it can vary throughout the system as the wall time of different nodes is not guaranteed to be equal.

Unless noted otherwise, we use event-time as our time measure for windowing, as this is commonly done in modern SPEs [1, 8].

With the notion of time, we automatically introduce the notion of order. When processing events, it could be expected that the event time of a tuple increases monotonically. However, due to the distributed nature of SPEs and their input streams, as well as unreliable networks, the order of the event stream is not always guaranteed. Thus, we make a distinction between *in-order* and *out-of-order* streams [42]. While in-order streams require no special attention, due to their logically correct timestamp order, out-of-order streams incur additional processing steps to ensure correct behavior if the order of tuples is relevant.

In out-of-order streams, watermarks are used to indicate that all events up to a certain timestamp have been processed [1]. Watermarks are special punctuations



(a) Tumbling window (b) Sliding window (c) Session window

Figure 8: Common window types.

in the stream, which are usually generated by the system itself. As the system cannot wait indefinitely for out-of-order events, it specifies a maximum allowed *lateness* after which an event is simply ignored. Periodically, the system introduces a watermark with a timestamp into the stream, which takes this lateness into account. Thus, if we see a watermark with timestamp 100 and have a maximum lateness of 10, we know that we have processed events with a timestamp greater than 110 and it is safe to discard any event older than 100 because it has become irrelevant.

There are also windows that use event counts as a boundary, e.g. "calculate the average price of the last ten stocks." We note here that count-based windows can also be seen as timed windows with a logical time that simply increases successively for consecutive events [1].

2.4.2 Window Types

We now present the most common window types and classify them in a formal model. Figure 8 shows the three most common windows: tumbling windows (Figure 8a), sliding windows (Figure 8b), and session windows (Figure 8c). Tumbling (or fixed) windows split the data stream into non-overlapping chunks of a certain length. Once a window ends, the next window automatically begins. A tumbling window is independent of the event data, it is still created if no data is present. An example query with a tumbling window could be: "calculate the average stock price of the last ten minutes."

A sliding window is defined by a window length and a window slide. The window slide determines when a new window starts after the previous one. If the window slide is smaller than the window length, windows overlap. In Figure 8b the length is ten seconds and the slide is five seconds, so a new window of length ten starts every five seconds, causing an overlap of five seconds between two consecutive windows. Tumbling windows are a special case of sliding windows, where the slide

is equal to the length. An example query for a sliding window could be: "calculate the average stock price of the last ten minutes, every five minutes."

Session windows group all events into a window until a certain period of time (window gap) has passed without seeing any events. In Figure 8c, the gap is defined as two seconds. The last event for the first window has a timestamp of eight. The next event has a time of twelve, so an inactive period of four seconds causes the first session to end. The length of a window is determined solely by the incoming data. If no data arrives, no sessions are created. If there are no sufficient gaps, a window can stretch indefinitely. An example of a session window could be: "calculate the average speed of a car before it becomes stationary for five minutes."

These three windows all deal with time as a boundary. As mentioned above, the window measure can also be count-based, i.e. using a special logical time and not physical time. Additionally, windows can be bound by special events, known as *punctuations*, that function as start and end markers. These punctuations are integrated into the regular event stream [60] and processed accordingly. These types of windows, however, are less common than the time-based versions as shown in Figure 8.

To deal with all these window types in a uniform manner, we classify them according to the terminology introduced by Li et al. [41] and refined by Traub et al. [58]. This classification is based on the context that the system needs to be able to determine a window start and/or end. Here, context simply describes information about events that arrived before (backward-context) or after (forward-context) the current event. The three groups are *context free*, *forward-context free*, and *forward-context aware*.

Context Free Windows (CF)
Context free windows do not require backward- or forward-context at all. The start and end times for a context free window can be determined simply by looking at the window definition. Tumbling and sliding windows are context free as all start- and endpoints can be calculated from the window length and slide if needed. For a sliding window with a length of ten seconds and a slide of five seconds, as seen in Figure 8b, we can determine the windows as Window 1 from second 0 to 10, Window 2 from second 5 to 15, Window 3 from second 10 to 20, Window 4 from second 15 to 25, and so on.

Forward-Context Free Windows (FCF)
Forward-context free windows require backward-context but no forward-context.

For an FCF window, we can determine the start- and endpoints of all windows up to point t, by having processed all events before to t. Thus, all information lies in the backward-context and none in the forward-context. Windows that are explicitly terminated with a punctuation are FCF. At time t, we have processed all punctuations up to t, so we can decide on all window starts and ends until t. For example, an FCF window query is "calculate the average stock price between two *trade-start* and *trade-end* messages".

Forward-Context Aware Windows (FCA)
A window is forward-context aware if it needs to process events after a timestamp t to determine all window start and end timestamps up to t. FCA windows require backward- and forward-context. The most common FCA window type is a session window. Once we have processed all events until t, we do not know if the event at t terminates the window or not. We need to wait for at least the duration of the session gap s to observe any further events. Once we received an event e with a timestamp greater than $t + s$ and no other event in the time from t to e, we can decide that the session terminated at t. An example of an FCA window is a session window, "calculate the average stock price until no stock prices have been seen for one hour".

In this work, we aim at supporting all possible window types. In Section 3, we discuss in detail how to support different windows in a distributed setting and which processing implications each type has.

2.5 Window Aggregation

Most modern streaming processing applications do not deal with just one query or window, they often run with dozens, hundreds, or even thousands of parallel queries and windows. As shown in Figure 1, even a single window can cause an overlap and redundant storage and computation in a naive implementation. This redundancy can quickly have a strong negative impact on the performance of a system [9, 45, 58]. To overcome the problem of unnecessary storage and computation, aggregate-sharing can be used. Aggregate-sharing describes the concept of reusing aggregations from previous, overlapping windows to avoid re-computing the same aggregation on the same data multiple times.

In this section, we introduce and characterize the different types of aggregate functions (see Section 2.5.1), followed by data aggregation concepts in Wireless Sensor Networks (see Section 2.5.2) and aggregate-sharing concepts in Stream Processing Engines (see Section 2.5.3).

2.5.1 Aggregate Functions

In this section, we introduce the different types of aggregation functions and categorize them according to their requirements for aggregate-sharing. For this, we first provide example aggregations and then sort them into the common groups *distributive*, *algebraic*, and *holistic*, as defined by Gray et al. [24]. We use the set of well-known aggregation functions `sum, average, min, max, count, median` to explain the three different aggregation groups. We also discuss which characteristics an aggregate function must provide in order to benefit from distributed aggregate computation.

Distributive Aggregate Functions
A `sum` function simply adds up the values of all elements.

$$\texttt{sum}(\{1, 2, 3, 4\}) = 1 + 2 + 3 + 4 = 10$$

One characteristic of the `sum` function is its distributive property, i.e. the ability to calculate the `sum` function on sub-parts of all data and then apply a merge function to the intermediate results. More formally, a function $F()$ is distributive if there exists a function $G()$ for which $F(X_{0..n}) = G(F(X_{0..i}), F(X_{i+1..n}))$ is true. For `sum`, the $G()$ is equal to the actual aggregation function $F()$. One example for which this does not necessarily hold is `count`. Applying `count` to intermediate results will not produce a correct result, but calling `sum` as $G()$ on the intermediate results will.

$$\texttt{sum}(\{1, 2, 3, 4\}) = \texttt{sum}(\{\texttt{sum}(\{1, 2\}), \texttt{sum}(\{3, 4\})\}) = \texttt{sum}(\{3, 7\}) = 10$$

This property can be applied recursively on any given sequence until there are only two elements left. We note here that any intermediate result can always be represented by a single element with a fixed size, e.g. using `sum` on a sequence of integers will always produce intermediate and final integer results.

Other common functions that share this property are `min`, `max`, and `count`.

$$\texttt{max}(\{5, 7, 2, 9\}) = \texttt{max}(\{\texttt{max}(\{5, 7\}), \texttt{max}(\{2, 9\})\}) = \texttt{max}(\{7, 9\}) = 9$$

We call functions with this property *distributive* aggregation functions.

Algebraic Aggregate Functions
Another class of aggregate functions are *algebraic* aggregation functions. A common algebraic aggregation is the `average` function. Formally, a function $F()$ is algebraic if there is a function $G()$, which returns an m-tuple, a function $H()$, which takes the m-tuple as an argument, and $F(X_{0..n}) = H(G(X_{0..i}), G(X_{i+1..n}))$ holds. For example, for the `average` function, $G()$ takes a sequence and calculates the `sum` and `count` as a 2-tuple. $H()$ then takes these 2-tuples, adds the partial sums and adds the partial counts. Then it divides the total sum by the total count to produce the average. To calculate `average({1, 2, 3, 4})`, we can split the sequence into $G(\{1, 2\})$ and $G(\{3, 4\})$. Here, $G()$ will produce the 2-tuples, defined as (sum, count), $(3, 2)$ and $(7, 2)$. $H()$ will then merge these to $(10, 4)$ and then divide 10 by 4 to produce 2.5 as the average. The key to algebraic functions is that they can be represented by a fixed-size m-tuple. Distributive functions are special algebraic functions with m-tuples of size one.

Holistic Aggregate Functions
All remaining aggregation functions, which cannot be represented by a fixed-size m-tuple, are grouped as *holistic* aggregate functions. A common example for a holistic function is `median`. For the `median`, we cannot split the sequence into subparts and merge their intermediate results. `median({1, 2, 3, 4, 5})` requires us to see all the data to determine 3 as the median value. Other common holistic functions are `quantiles` and `mostFrequent` (or `rank`).

Decomposable Functions
With regard to distributed aggregation, we additionally use the terminology introduced by Jesus et al. [35]. They distinguish between decomposable, self-decomposable, and by elimination, non-decomposable aggregation functions. Here, decomposable refers to the characteristic of being able to split a function to work on subsets of the data. Distributed aggregation requires aggregate functions to be decomposable, otherwise there is no benefit in the distribution of the data, as all of it needs to be gathered and processed together. Holistic functions map trivially to non-decomposable functions, as they cannot be split. A slightly different distinction is made between self-decomposable/decomposable and distributive/algebraic. Algebraic functions are, by our definition above, decomposable. We are able to split the input sequence into subsets, perform the aggregation on them, and then merge the intermediate results. However, not all distributive functions are self-decomposable. As mentioned above, most distributive functions can be split recursively, with $F() = H()$, e.g. `sum` or `max`. Functions with this equality-property are defined as self-decomposable, the rest is simply decomposable. As mentioned above, `count` is distributive but not self-decomposable.

For the remainder of this work, we do not require a strict distinction between decomposable and its special form self-decomposable. It is sufficient information to know if a function can be split or not. We simply refer to decomposable functions for all algebraic and distributive aggregations, and to non-decomposable functions for holistic aggregations.

2.5.2 Data Aggregation in Wireless Sensor Networks

In systems with thousands of sensors or more, the data volume that has to be sent via the network becomes a major influence on the performance of the entire system. If not taken into consideration, the network cost can quickly become intolerably high in low-bandwidth settings. In order to avoid such network bottlenecks, Wireless Sensor Networks (WSN) often rely on data aggregation while passing events through the system. As described above, we use the hierarchical or tree-like structured approach for our node communication. Thus, we only look into aggregate techniques applicable to hierarchical WSNs. For a complete overview of data aggregation in hierarchical and non-hierarchical WSNs, we refer the reader to previous research [14, 35].

At this point, we discuss existing WSN data aggregation approaches along two axes: aggregation on intermediate nodes vs. aggregation at the sink node and exact vs. approximate aggregation results. We first discuss the location of the aggregation and their respective advantages, followed by a comparison of exact and approximate methods. We also note here that data aggregation in WSNs depends heavily on its exact setup and function. A network with tens of thousands of sensors that each produce multiple events per second will require different handling than a network with one hundred sensors, each emitting one event every ten seconds. This needs to be taken into account when looking at the design decisions for distributed aggregation. In a high-volume system, it is more critical to reduce the network bandwidth than in smaller networks.

Aggregation Location
One approach to reduce network bandwidth is by aggregating data on intermediate nodes along the path from the individual sensor to the sink. There are many are different network structures that support this kind of path-based aggregation. We now present two main structures for path-based aggregation. First, the nodes can be ordered hierarchically in a tree-like setup, e.g. as proposed in *TAG* by Madden et al. [45]. In this tree, data is sent from nodes to parent nodes and aggregated there. Each intermediate tree node is thus responsible for all values from its subtree. The sink is the root of the tree and can compute the final aggregate from

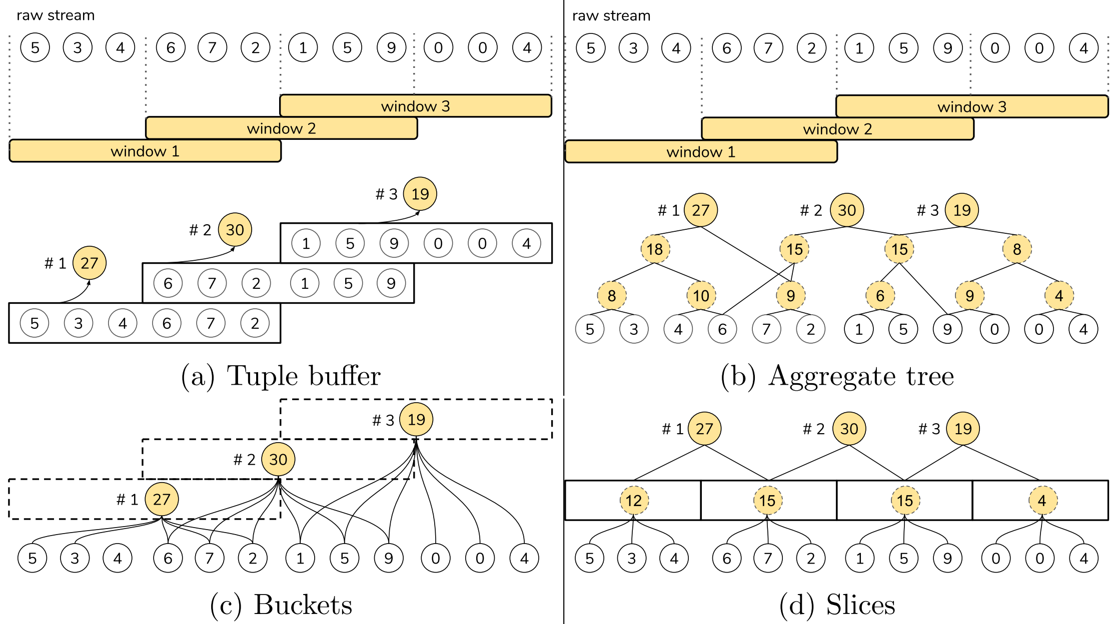
all intermediate values. A central weakness of such tree-based approaches is fault tolerance. If a single node along the path fails, the values of the entire sub-tree are lost. However, the main advantage is low the number of hops that all events have to take and the distribution of computational cost between nodes. In balanced trees, the number of hops is logarithmic in the number of nodes and intermediate nodes need to receive only a few messages compared to a single sink node receiving all events.

Another approach to path-based aggregation is along a global chain of nodes, e.g. as proposed by Lindsey et al. in PEGASIS [43]. The idea of a node-chain is to pass the aggregate from node to node until each node has added its value and the combined aggregate has reached the sink node. This way, the aggregation load is evenly distributed between all nodes and if a node fails, the message can simply be sent to the next node in the chain. However, this comes at the cost of a high number of network hops, as each node needs to have its turn. Thus, the number of hops grows linearly with the number of nodes. Also, this approach requires a global ordering of nodes in order to maintain a minimum number of messages and computing a globally ordered chain can be very expensive in networks with many nodes.

We refer to previous work [14, 26, 35, 46, 50] for a more detailed overview for path-based aggregation algorithms in WSNs. As our goal is to minimize the amount of data that needs to be sent at each network hop with a low latency, we focus more on a tree-based approach where we group data at intermediate nodes. Also, a chain-like approach requires global state at each node, which comes with a high cost if more information than the network structure is needed. Fault-tolerance aspects are not within the scope of this thesis and are left for future work.

Aggregation Accuracy
An orthogonal method to reduce the network load is by using approximate data aggregation instead of exact values. While most approaches work on exact values, some use cases allow for an approximate aggregation. The main idea behind approximate aggregation algorithms in WSNs is to provide flexibility and fault-tolerance at the cost of exact results. When nodes enter and leave or fail, values are added or lost. Without synchronization, exactly-once processing becomes increasingly harder with more nodes. By simply broadcasting its own value and letting approximation data structures, such as Sketches [13], do the partially duplicate-insensitive aggregation on other nodes, the system becomes highly flexible to node failures. We refer to previous work [13, 35, 50] for a broad overview of existing solutions.

In contrast, most hierarchical algorithms like *TAG* and *PEGASIS* work on exact



(a) Tuple buffer (b) Aggregate tree (c) Buckets (d) Slices

Figure 9: Different aggregate-sharing approaches in SPEs.

values. Our aim is to provide exact aggregation results for arbitrary aggregation functions, so in this work we focus on algorithms for exact data aggregation.

2.5.3 Aggregate-Sharing in Stream Processing Engines

Modern Stream Processing Engines often process hundreds of queries with varying window types and lengths concurrently. As mentioned above, even single windows can lead to significant overlaps and require redundant storage and computation (see Figure 1). The widely used SPE Apache Flink [8] deals with this overlap by re-computing aggregates for each window, leading to degraded performance in, e.g. a long sliding window with a short slide [19]. Thus, there is a need for a more efficient handling of window aggregation [17, 18]. In this section, we present common approaches to improve the performance of overlapping windows by applying aggregate-sharing. Aggregate-sharing refers to the concept of reusing previously computed aggregate values for overlapping parts of windows. We first present the naive approach using a simple tuple buffer, followed by buckets, aggregate trees, and slices.

Tuple Buffer
Tuple buffers are the naive way to process window aggregations. The idea behind them is to create a new buffer for each window and then add each incoming event to all of the buffers that it belongs to. A tuple buffer does not share any intermediate aggregation results. An illustration of a tuple buffer can be seen in Figure 9a. For systems with no or only a few overlapping windows, this approach is feasible, as there is no or only minimal overhead. However, increasing window overlaps negatively impact the system's performance, as aggregations on the same data need to be computed multiple times. Also, the latency of a tuple buffer, in general, is rather high because all computation is performed at the window end. The memory requirements of tuple buffers increase linearly with the length of the window. A window of ten-minute length and an input stream of 1000 events per second will lead to a buffer size of 600000 events. If this window overlaps for five minutes, we store 300000 events redundantly. Processing out-of-order events also has a high impact on the tuple buffer performance, as the entire buffer needs to be copied with the event placed at the correct position in the buffer.

Buckets
To extend tuple buffers to be order independent, Li et al. propose Window-ID (WID) [41, 42]. WID uses a *bucket* operator to add a set of window ids to each tuple, according to which windows a tuple belongs to. The aggregation operator then uses these window ids, in addition to the aggregation operation, to compute partial aggregates for all given windows. Thus, the order of the tuples is irrelevant to the aggregation. WID uses stream punctuations to explicitly end windows and inform the aggregation operator to emit the final aggregate. This process leads to a much lower latency than plain tuple buffers, as the final aggregate has already been computed at the window end. However, there is no aggregate-sharing for overlapping windows. The aggregation is performed per window, leading to redundant computation for single events, as can be seen in Figure 9c. This has a high impact on the throughput of a system with many overlapping windows. Depending on the aggregation function and window, events need to be held or can be discarded after adding them to the aggregate. Thus, we distinguish between tuple buckets and aggregate buckets. Aggregate buckets add each tuple to the aggregate, store some additional metadata (e.g. window start and end timestamps), and then discard the event. This can be used, e.g. in a tumbling window with the `sum` aggregate function. Tuple buckets, on the other hand, store the individual events as no partial aggregate can be computed. For example, holistic aggregate functions, such as `median` require this approach as they need the entirety of the data to aggregate correctly. This leads to redundant storage for overlapping windows.

Aggregate Trees

Another approach to efficiently compute aggregates on data streams is tree-based aggregation. Common algorithms here are B-Int [3], as well as FlatFAT [55]. FlatFAT builds a binary tree on top of the raw event stream, i.e. each event is a leaf in the tree, whereas B-Int groups events into disjoint intervals which are then aggregated in a tree. The intervals for B-Int can be split arbitrarily, however, the smallest intervals must always align with the window sizes. Each level of the tree stores the aggregates of its children and ended windows can use the partial aggregates from the tree to compute the final aggregation. The latency of aggregate trees is lower than that of tuple buffers, as only a few aggregations need to be performed on pre-aggregates for a given window compared to aggregating all individual events. Windows can generally also reuse the aggregates of overlapping sub-trees, avoiding re-computation. Out-of-order events cause $O(log\ n)$ updates (with n leaves) in the tree in both algorithms, as a leaf value is updated, as well as a potential expensive memory copy in the list of leaves. In-order events can be processed without that update cost and generally require $O(n)$ additional space for the tree on top of the events. An illustration of the aggregate tree from FlatFAT can be seen in Figure 9b.

Slices

The last group of methods to efficiently aggregate stream data does this via *slices*. A slice is a finite chunk of the event stream and all slices are disjoint [39, 40, 58]. Each slice computes a running aggregate for incoming events and a window then performs the final aggregation on only the pre-aggregated slice values. This leads to a similarly high throughput and low latency as with aggregate trees, for the same reason. One main advantage of slices over the other approaches is that each event is processed only once, as it belongs to exactly one slice and there is no overlap between slices. This is depicted in Figure 9d, where each tuple belongs to only one slice. Analogously to *buckets*, slices can either contain only a single aggregated value or also store the raw events, depending on the window type and aggregation function. Thus, in many cases, slices do not need to store raw events which leads to a drastic reduction in memory consumption.

We focus on slicing approaches for this work, as they provide a high throughput and low latency as well as a low memory footprint in many cases. We also make use of the semantics of an event belonging to a single slice with no overlaps to avoid sending events through the network multiple times for a distributed aggregation. Also, recent research by Traub et al. [58] shows that it is possible to cover all possible window types with a general slicing approach.

2.6 General Stream Slicing

To efficiently aggregate a stream of raw events when it enters the system, we make use of *general stream slicing* [58]. It provides good latency and throughput characteristics, and it produces non-overlapping slices that we leverage in our distributed aggregation. We note, however, that our distributed aggregation approach is theoretically agnostic to the exact slicing algorithm used, as long as correct non-overlapping slices and windows are produced. It can also be built on top of alternative slicing approaches, e.g. *Cutty* [9], *Panes* [40], or *Pairs* [39]. Yet, general stream slicing is the most sophisticated and feature-rich algorithm, so we choose it over the alternatives. We discuss the shortcomings of the alternative slicing designs in Section 6. In this section, we describe the main mechanics of general stream slicing. We refer to the original paper for more details on this approach.

Event Storage
As mentioned above, depending on the type of window and aggregation function, it may be required to store all raw events instead of just the partial aggregate per slice. The first criterion is based on the order of the stream. If it is in-order, in nearly all cases the raw events can be discarded after processing them for the partial aggregate. For in-order streams, only forward-context aware windows need to store all events because future incoming events can influence the window start and end times, making it impossible to pre-compute a partial aggregate. Session windows make an exception, as they are FCA but do not need re-computation. This is due to the fact that session windows never shift over tuples but only ever increase in size [59]. If a stream is out-of-order, there are a few more factors that determine the storage of raw events. If the aggregation function is non-commutative ($\exists x, y : x \oplus y \neq y \oplus x$), i.e. the order of the events is relevant, the events need to be stored to allow for correct re-aggregation. If the order is irrelevant, but the window FCA (excl. session windows), the tuples need to be kept for the same reason as above. Finally, if a window is context free, it depends on whether it is count-based or not. Count-based windows need to store events, as the window may shift across the stream, removing events from the front of the window. Time-based windows, on the other hand, can discard all tuples after processing.

Slice Management
General stream slicing uses three main operations on slices, *i*) `merge`, *ii*) `split`, and *iii*) `update`. The `merge` operation takes two slices and merges them into one. For this, the end timestamp of the earlier slice is updated to the end timestamp of the later, the partial aggregates and/or raw events of both slices are combined, and the later slice is discarded. The combination of state can be either constant

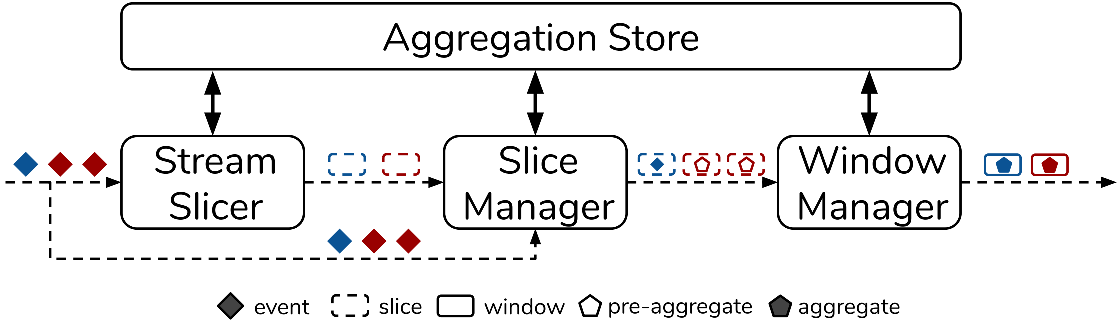


Figure 10: General stream slicing process data flow.

for decomposable functions, or more costly for holistic ones. The `split` operation performs the opposite by taking one slice and splitting it into two at a given timestamp t. The end timestamp of the existing slice is set to t, the new slice starts at $t + 1$ and ends at the old end of the original slice. Then, the partial aggregates for both slices are re-computed. A `split` is only possible if the raw events were stored, otherwise, a re-computation is not possible. All out-of-order context aware streams require raw event storage, as the backward context may be altered by an incoming late event. The `update` operation can add or remove a tuple and it can update the slice's start and end timestamps (or other metadata in general). Adding an event simply adds its value to the partial aggregate for in-order and order-insensitive out-of-order events. For order-sensitive aggregations, it triggers a complete re-computation. Removing an event causes a complete re-computation of the pre-aggregate unless the aggregation function is invertible (e.g. `sum`), in which case the value of the event-to-remove is simply removed from the aggregate.

Event Processing

We now describe the flow of a single event that enters the system. The general components are shown in Figure 10. The Aggregation Store is shared between all other components to correctly create and update slices as well as compute the final window aggregate.

The *Stream Slicer* creates new slices, when needed, for incoming events. Whenever a window starts or ends, the Stream Slicer creates a new slice. It stores the timestamp of the next window end and compares it to each event. If an event's timestamp is later that the window's end, a new slice is created. This method is very efficient as it simply needs to compare two timestamps per event. The next window end is calculated as the earliest end bound of all present windows. Out-of-order tuples are simply passed on, as the slice they belong to has already been

created. As calculating the next end of each window is heavily dependent on the window type, the authors provide a simple interface for getting the next window edge, which takes in the current timestamp. For CF windows of length l, this can be a simple static calculation $timestamp + l - (timestamp \bmod l)$. For FCF and FCA this also requires the context of the current window and can be different after each event, depending on whether that event updated the context in a significant way. A session window, e.g. updates its window end after each new event to the event's timestamp plus the session window gap. We note that a window end can initially be undefined, thus, making the next slice infinite. However, as soon as a window edge can be computed from the context, the slice can be updated to reflect this.

In the *Slice Manager*, the incoming event can trigger an `update`, `split`, or `merge` operation. Context free windows never cause `splits` or `merges`, as their bounds are known ahead of time and don't change. The Slice Manager makes sure that the slice bounds always align with window start and/or end bounds, in order to guarantee the minimum possible number of slices [9]. If the stream is context aware or out-of-order, the Slice Manager checks if the event changes existing context, and thus window start or end timestamps, and performs the according `merge` and `split` operation to create the required slices matching the updated window bounds. As the slices are always created at the window edges, they are guaranteed to fit into updated windows if a window is extended and can be split accordingly if the window is shortened. Should the updated context cause new window edges, the Slice Manager informs the Window Manager about the updated bounds so complete windows can be triggered. Finally, the Slice Manager adds the event to the correct slice, according to its timestamp, with an `update` performing the aggregate computation as described above. This can lead to simple incremental aggregate updates or to a complete slice aggregate re-computation for non-commutative aggregation functions.

As the last step, the *Window Manager* is responsible for calculating the final window aggregate from the associated slices. It does this by waiting for watermarks in the stream. Once a watermark reaches the Window Manager, it can trigger all windows with an end bound smaller than the watermark. As all windows have clearly defined window ends accessible through the interface mentioned above, the Window Manager can easily determine which windows end when. For in-order streams, this watermark is implicitly given by the timestamp of the current event, as there are no late events. In out-of-order streams, the Window Manager waits for an explicit watermark event. The final aggregate for a window is computed by collecting all slices belonging to that window and either aggregating the slice's pre-aggregates for decomposable aggregation functions or merging all raw events

and computing the result on the entire event set for non-decomposable functions.

Also, *general stream slicing* allows the user to define custom aggregation functions based on a three method interface, `lift`, `combine`, and `lower`. We discuss these three methods in Section 3.1, as we aim to support arbitrary windows in this work.

2.7 Summary

In this section, we presented the requirements and background for our distributed aggregation solution. For this, we first discussed at Wireless Sensor Networks (WSN) and Stream Processing Engines (SPE), as well as their applicability to Fog computing scenarios. We then discussed windowing techniques to perform analytical queries on infinite data streams, followed by methods to efficiently compute aggregates on such windows. Finally, we presented at a state-of-the-art stream slicing approach as a way to perform local data aggregation with low latency and high throughput characteristics.

In this work, we focus on combining aspects from both Wireless Sensor Networks and Stream Processing Engines, by applying the sensors' efficient tree-based data transmission combined with SPEs' complex windowing and aggregation logic. In the following section, we present our distributed data aggregation solution and discuss how to achieve this combination efficiently.

3 Distributed Data Aggregation Concepts

Efficiently processing streaming data is becoming increasingly hard with the rise of the Internet of Things and the mass of senor data it entails. Sensor networks produce millions of events per second in locations all across the globe. Thus, processing all data in a central cloud environment comes with a high network cost for raw data transfer. In order to avoid this high cost, previous research shows methods to aggregate data along the path from sensor to cloud. In Wireless Sensor Networks, hierarchical network structures are often used for this [43, 45]. However, these solutions support only basic aggregation functions and simple window types. On the other hand, modern Stream Processing Engines allow the user to define complex windowing and aggregation logic [8, 70] but do not support in-network data aggregation. In order to bridge this gap between complex business logic and efficient data transmission, we present Disco, a stream processing framework for efficient in-network data aggregation on arbitrary windows.

The basis of Disco is the distributed aggregation of complex window types, so in this section we first introduce some definitions to describe the necessary actions (see Section 3.1), followed by detailed merging strategies for each of the different window classes (see Section 3.2) and aggregation functions (see Section 3.3).

3.1 Definitions

In this section, we introduce operations that describe how windows and their corresponding aggregate values can be merged and combined. To describe the aggregation steps, we use the terms `lift`, `combine`, and `lower` introduced by Tangwongsan et al. [55]. For these steps, we also introduce three types for an aggregation, the input type `IN`, the output type `OUT`, and the partial type `PARTIAL`. Here, `IN` denotes the type of the single value that should be added to the aggregation, e.g. an integer value or a float. The `OUT` type describes the value that the function returns as the final result. The `PARTIAL` type is used to describe an intermediate representation of an aggregate. We need this if the aggregation function converts the input to a different type, e.g. integer to float. We show these types for the aggregation functions `sum`, `median`, and `count`. For the `sum` aggregation `sum`($\{2, 4, 6\}$), all three types `IN`, `PARTIAL`, and `OUT` are integer values. For the `median` function `median`($\{2, 4, 6\}$), the types are not equal. `IN` is an integer, `PARTIAL` is a list of integers, and `OUT` is an integer again. For `count`, we possibly need to convert between type domains, e.g. the `IN` type could be string but the `PARTIAL` and `OUT` types are integers. These three steps, `lift`, `combine`, and `lower` and types, `IN`, `PARTIAL`, and `OUT` can be used to describe any aggregation function.

`lift(IN` x`)` $\rightarrow$ `PARTIAL`: `lift` converts the input value of type `IN` to the required intermediate state representation of type `PARTIAL`. For example, `lift` for `sum` takes a value x and simply returns x unchanged. For `avg`, it takes the value x and returns the tuple $(x, 1)$ and for `median` it would take x and return the single-element list $\{x\}$.

`combine(PARTIAL` $agg1$`, PARTIAL` $agg2$`)` $\rightarrow$ `PARTIAL`: In many cases, this describes the actual aggregation step, which takes two `PARTIAL` values and combines them into one value, also of type `PARTIAL`. For `sum`, this would simply add $agg1$ and $agg2$. For `avg`, `combine` would return the tuple ($agg1.sum$ + $agg2.sum$, $agg1.count$ + $agg2.count$). For `median`, this is not the actual aggregation step but simply the merging partial states, so it would return the merged list $\{agg1_0, ..., agg1_k$, $agg2_0, ..., agg2_n\}$. For all decomposable aggregations, `combine` performs a partial aggrgation towards the final result.

`lower(PARTIAL` agg`)` $\rightarrow$ `OUT`: `lower` is the last aggregation step and returns the final aggregation result. To this end, it takes the final intermediate representation and converts it to the output type. For the `sum` aggregation, `lower` would simply take agg and return it unchanged. For `avg`, it would divide the $agg.sum$ by $agg.count$ and for `median` it would sort the agg list and return the middle element.

We define an aggregation window w as a triple $(start, end, state)$, where $start$ is the value of the start-bound, end is the value of the end-bound, and $state$ is the intermediate representation `PARTIAL` of the given aggregation function. We note here that $start$ and end can take on the form of any window measure.

We define the binary window-merge operation $\cup$ in order to merge two windows into a new one.

$$\begin{aligned} w &= w_1 \cup w_2, \text{where} \\ w.start &= min(w_1.start, w_2.start), \\ w.end &= max(w_1.end, w_2.end), \\ w.state &= \texttt{combine}(w_1.state, w_2.state) \end{aligned}$$

Based on these operations, we discuss how distributed windows and aggregation functions are correctly handled. To this end, we pick up on the formal window types *context free*, *forward-context free*, and *forward-context aware*, as introduced in Section 2.4.2, and present how they can be merged in a distributed setting (see Section 3.2). Finally, we discuss the three classes of aggregation functions (*distributive*, *algebraic*, *holistic*) in Section 3.3 and discuss how they influence the ability to create partial aggregates on a subset of the data on different nodes.

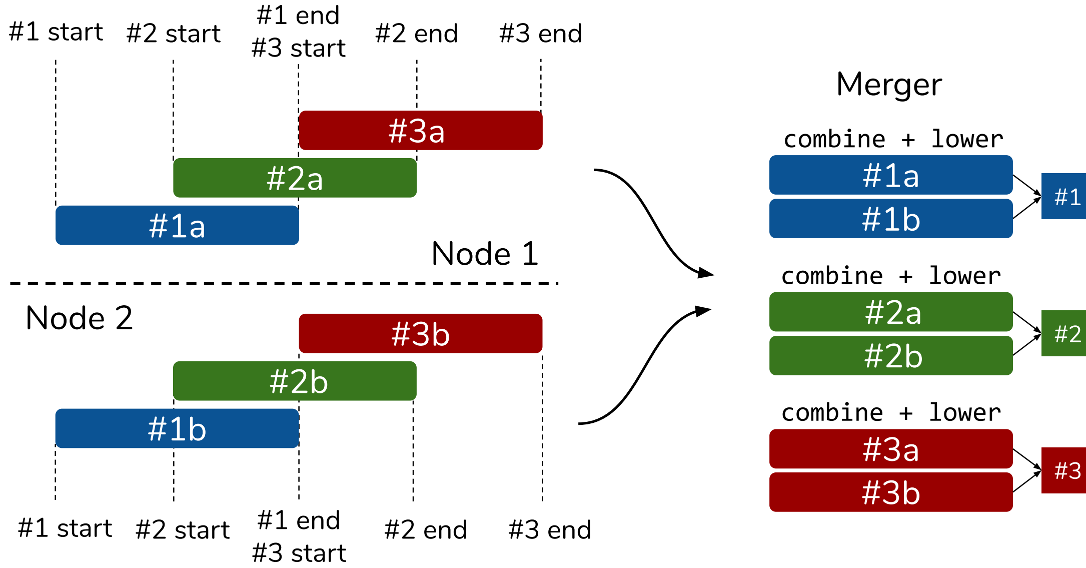


Figure 11: Merging distributed context free windows.

3.2 Window Types

Depending on the window characteristics (e.g. type and measure), we select the optimal window creation and merging strategy. In particular, we differentiate between the three window types, presented in Section 2.4.2, *context free*, *forward-context free*, and *forward-context aware*, as well as the window measures *time*, *count*, *punctuation*. For simplicity, we assume there are always k independent nodes producing windows that need to be merged by one "merger" instance to produce a global result. We also explain these concepts without the use of keys in events. Usually, an application would perform certain aggregations by key, e.g. the click by a certain user (user id = key). Without loss of generality, the explanations below all work on a single key for any given window. We can extend this to multiple keys by applying the concepts to each unique key.

Context Free Windows (CF)
Context Free (CF) windows are the most straightforward to deal with in a distributed setting. We show this for a simple sliding window on two nodes in Figure 11. The bounds of each window can be computed statically and ahead-of-time, thus making them equal on each node. As each node produces windows with the same bounds, merging the windows becomes trivial. For each logical window w with $w.start = x$ and $w.end = y$, we simply collect all k matching intermediate windows out of all windows W, $\{w_i \in W \mid w_i.start = x \wedge w_i.end = y\}$ and merge

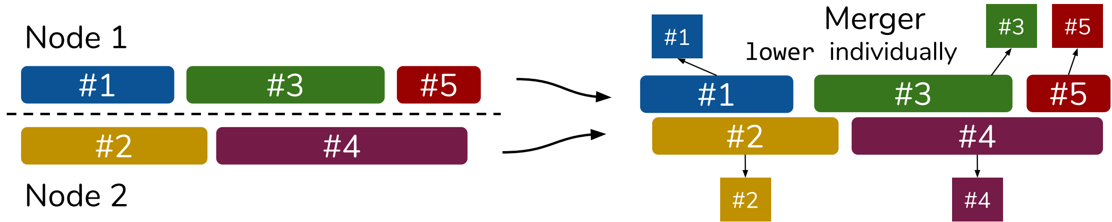


Figure 12: No window merging for local FCF windows.

them into a new global result window $w = w_1 \cup w_2 \cup ... \cup w_k$. The result is then calculated by calling `lower`$(w.state)$. An example of such a distributed context free window is a smart home setting in which the smart thermometer company wishes to know the average temperature of all houses for the last hour.

A CF window can also be interpreted locally, i.e. there is no window merging as the windows produced by each individual node are viewed as final windows. This makes sense in cases where there are no overlapping keys on different nodes and the query is interested in aggregation on single keys only. A resulting local window w is passed to the window merger, which simply emits w and `lower`$(w.state)$ as a global result. In that case, we still benefit from distributed aggregation as we do not need to transfer raw data past the first node in the system and we naturally distribute the cost of computation across all nodes.

Forward-Context Free Windows (FCF)

There are two possible ways to define FCF windows in a distributed setting, *i)* as local windows or as *ii)* global windows. The decision of which to choose is application dependent and needs to be decided by the user. We explain the two types with the example of punctuation-based windows. As there is no one large stream with a single order in which a window-end punctuation can be ingested, we need to distinguish between a local window-end and a global window-end. A local window-end is a marker denoting the end of a window in a single stream, whereas the global window-end marks the end of a window in all streams combined. We note here that the global interpretations are all application- and data-specific. It is not always possible to simply interpret an arbitrary, logically-unified stream of data in multiple disjoint sub-streams. As soon as the window requires a global ordering of events, we cannot profit from distributed aggregation. For the cases in which this is possible, due to knowledge of the underlying data and application, we present three interpretations.

For a *local window*, a window-end is ingested into a single stream, thus bounding an existing window only on the node processing that exact stream. The result-

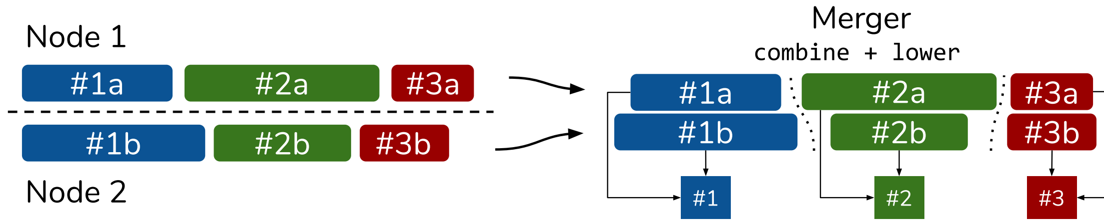


Figure 13: Merging FCF windows by processing order.

ing window w is then passed to the window merger, which simply emits w and `lower`$(w.state)$ as a global result. This is shown in Figure 12, where each window is processed individually. An example of such a local window is again a smart home setting in which the company wants to know the average temperature of certain houses while there heating is activated. Once it is turned off, the window is marked as complete and can be emitted. Similar to CF windows, this interpretation saves network costs and distributes the cost of computation across all nodes.

A *global window*, on the other hand, requires some kind of equivalence between punctuations in order to compare them and merge accordingly. Given two streams S_1 and S_2, and their respective punctuations p_1, p_2, p_3 and p_4, p_5, p_6, where p_t denotes a punctuation with the id t, there are three different approaches to interpret these punctuations. The first of which is when the system controls the ingest of these punctuations. In that case, the punctuations could be ingested at the same time or with the same logical id on all nodes. For this example, we assume that a punctuation marks the end of an existing window and the start of a new window. This would lead to the same outcome as for CF windows, where we simply collect and merge all windows with $w.start = x$ and $w.end = y$. In a smart home setting, this could be represented by the energy provider wanting to aggregate data for certain periods of time throughout all homes. The punctuation would be inserted at a certain fixed time by the system, which is then identical on all devices. This can be generalized to all FCF windows in the sense that application- or system-logic provides each window with an equivalent backward context from which the window bounds can be computed. However, this is not necessarily possible. In that case, an alternative interpretation may be applicable or else there is no benefit in a distributed aggregation and the raw data needs to be collected at a central node.

The second *global window* interpretation is the equivalence based on the absolute count position of a punctuation, i.e. $p_1 \equiv p_4$, $p_2 \equiv p_5$, and $p_3 \equiv p_6$. Each node creates its windows locally based on the punctuations that it receives and

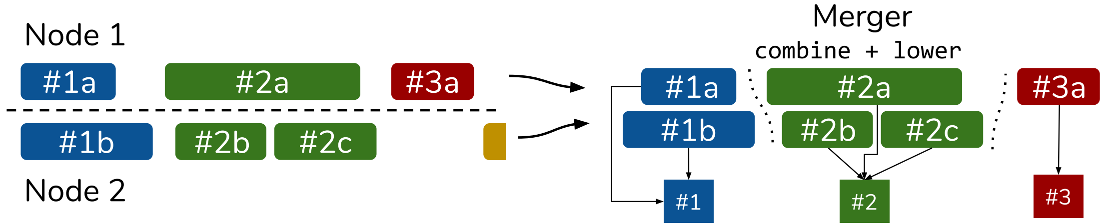


Figure 14: Merging FCF windows by global gaps.

we can then collect and merge windows accordingly. We show this in Figure 13, where the bounds of the individual windows are not aligned but the windows are still combined based on their order of processing. For the ith logical window we can collect the k matching windows from the nodes' respective window sets $W_1, ..., W_k$ at position i. They can then be merged into a new global result window $w = w_1 \cup w_2 \cup ... \cup w_k$. This order-based merging can be generalized to all FCF windows in the sense that we simply interpret the ith window that each node produces as equivalent to all other i^{th} windows. An example of such order-based merging is location tracking for people taking a train via a route provided by a navigation app. There are three distinct stages in the journey: getting to the train, actually taking the train, and reaching your final destination after leaving the train. If the navigation app is now interested in the average speed of the person at each stage, this can be modeled by an order-based merging. This results in three windows, one for events while getting to the train, one window while on the train, and one after leaving the train. Generally, this could be applied to any stream where there is a fixed number of distinct sub-streams that are logically equivalent across the streams. We note that this can produce windows with a large difference between their start and end bound. For example, the stream S_1 introduces many punctuations with little time between them and thus many windows are created. In the same time period, S_2 has only resulted in one window, so now the bounds of the result window are far apart and will overlap with the next window, as the second window of S_1 starts before the result window ended.

The third *global window* idea is based on explicit start and end punctuations in each stream. We then have periods of active windows and periods of inactive windows on the different nodes. Each node can produce and emit its windows according to the received punctuations. We can then collect and merge windows based on overlapping periods of activity. For this, we simply collect the windows emitted by the k nodes and check if there is an overlap with any previously collected windows. Should a window w_{new} overlap with a window w_{old}, it is merged to produce a new window $w = w_{old} \cup w_{new}$ in their place. Two windows overlap if, and only if $w_{old}.start \leq w_{new}.end \land w_{new}.start \leq w_{old}.end$. Intuitively, this is defined as "the

start of each window must be before the end of the other". An arriving window can also merge multiple windows, e.g. if it is a very long window or it closes the gap between two currently non-overlapping windows. We show this in Figure 14, where all overlapping windows are merged together. Even though there are two windows for #2 on Node 2, they are still combined with #2a, as their bounds overlap.

We define a global gap as a period of inactivity in all streams. A timestamp t is *active* in the set of all windows W if $\exists w \in W \ : \ w.start < t \wedge w.end > t$. A timestamp t is thus *inactive* if it is not active, i.e. there are no windows spanning across t. A period of inactivity is defined by an inactive timestamp t_{start} and an inactive timestamp t_{end} between which there are no active timestamps, i.e. (t_{start}, t_{end}) is inactive $\iff \forall t \in \{t_{start}, ..., t_{end}\} \ : \ t$ is inactive. For any collected and merged window w_m, we can now emit its result if there exists a period of inactivity directly following the window's end, i.e. $w_m.end + 1$ in inactive. The $+1$ here refers to the smallest supported unit of the given window measure, e.g. $+1$ nanosecond or millisecond or $+1$ event in a count-based measure. To determine if there is a period of inactivity directly following a given window, we must collect the windows of all k nodes that started after $w_m.end$ to ensure that there are no more active windows crossing this timestamp. In Section 4.3 we discuss the practical implications of waiting for all of the following windows and how this can be avoided in certain cases to reduce latency.

This interpretation can be generalized to all FCF windows by ensuring that there are global "gaps" in all streams. It is logically equivalent to having one large stream with all data and all gaps in it. This interpretation, thus, only makes sense if the data supports these gaps and the application requires them. If there is no global gap, there can be no global window and no final aggregation result. We also note that this is possible regardless of the window measure. A gap can be real time passing, it can be logical time passing, and it can be inactivity between explicit start and end punctuations. This even holds for count-based windows, as they can be interpreted as events with a logical time. To simulate a gap in a count-based window, e.g. the system could process dummy events that increase the count by the required gap, which is a crude mechanism to progress "time". An example of such a global gap-based window could be the temperature data of an office building, where the company wants to know how warm each office is for each day. The windows are started once people enter their office in the morning and triggered when they leave in the evening. The exact timing of the windows is not relevant, but they all overlap for a given workday.

However, there are queries for which none of these interpretations hold. For these windows, there is no benefit in processing the streams distributed and they need to be collected at a single node to be processed correctly. The interpretations all require knowledge of the application and data to decide which is appropriate. For example, choosing a window-merge based on processing order only makes sense if the windows produced by the individual nodes cover logically connected events. If the user cannot make any assumptions about the data it needs to be processed centrally.

A user-defined windowing function can be used to caputre complex logic relying on global backward-context. A complex custom window for which none of the interpretations hold is, e.g. "calculate the average stock price until the last five stock prices were monotonically increasing". This type of query is possible, albeit rather constructed. The window start is always determined by the end of the previous window and is active until we find a monotonically increasing sub-sequence in the backward-context. If this is not processed centrally, an incorrect result can be emitted. Let node 1 see the stock prices \$1 at $t = 1$, \$4 at $t = 4$, \$5 at $t = 6$, \$7 at $t = 8$, and \$8 at $t = 10$. Node 2 sees \$1 at $t = 2$, \$2 at $t = 3$, \$3 at $t = 5$, \$10 at $t = 7$, and \$9 at $t = 9$. Now, node 1 would trigger its window, as all its values are monotonically increasing. But this produces a false positive, as the stock prices are not really monotonically increasing. At $t = 5$, the stock price is \$3 but before, at $t = 4$, the price was \$4, so the monotonicity is broken. Only a global ordering can produce correct results in such a case. However, this limits most applications only theoretically, as very few use complex custom windowing logic but rely on the SPE's built-in common types.

Forward-Context Aware Windows (FCA)
Analog to FCF windows, FCA windows can be seen as local or global. The handling of local FCA windows is identical to that of FCF windows, with the same advantages of reduced network cost and even load distribution. The window w is created locally on a node, passed to the merger, and then simply emitted with $\texttt{lower}(w.state)$.

Again, global windows are only possible if the windows do not require a global ordering of events. If this is not required, two of the three global interpretations of FCF windows can also be applied to FCA windows. The first interpretation, based on the equivalent start and end bounds cannot be applied to FCA windows. This approach relies on explicit end bounds, which cannot be provided for FCA windows. If an explicit end bound can be set, it is by definition not an FCA window, as there is no need to forward-context. For the second interpretation, we can still process the windows based on the order they appear in. This is generally

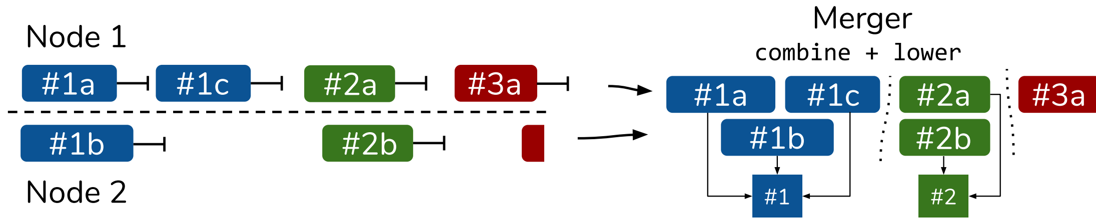


Figure 15: Merging distributed session windows.

applicable to all types of windows but requires knowledge of the underlying data to be able to make this assumption. In that case, it is irrelevant how the window was created as long as it can be logically merged with others. The third interpretation also holds for FCA windows, as there can be global gaps between windows regardless of their creation. However, again these interpretations all depend heavily on the knowledge of the underlying data and cannot be applied generally.

Session windows represent a special case of FCA windows and require a slightly different gap-based interpretation. A session window can be created locally and handled analogously to the third interpretation of FCF windows above, with explicit gaps in all streams. By nature, session windows already come with the notion of active periods and inactive periods. If there is no global gap in all of the streams, there is no use for a session window, as there would be no gap in a single merged stream either. Thus, we can assume that if a session window is chosen, there is a global gap. On one node, if there is no event for at least the length of the session gap, a session window is triggered and sent to the merger. The merger can then compare the incoming window to all previously collected, and possibly merged windows. In contrast to the overlap for FCF windows, a session window also needs to take the session gap into account. If a new window falls into a previous gap, the windows must be merged, even if there is no explicit overlap in the start and end bounds. We show this in Figure 15, where the second window #1c of Node 1 is only merged with the other two because it overlaps with the session gap from Node 2's #1b window.

We extend the overlap equation from FCF windows to include this session gap, $w_{old}.start \leq w_{new}.end + gap \land w_{new}.start \leq w_{old}.end + gap$. The gap is equal for both windows, as they are part of the same window definition. This merge can also span multiple existing windows, e.g. if the session is very long or it bridges the gap between two previously non-overlapping windows. The period of inactivity that decides whether a session window w is complete also needs to include the session gap. Compared to FCF windows, the required period of inactivity is now defined as $(w.end, w.end + gap)$. Again, all following windows must be collected in order to correctly determine whether the current session window is complete, which can cause a high latency if not handled correctly.

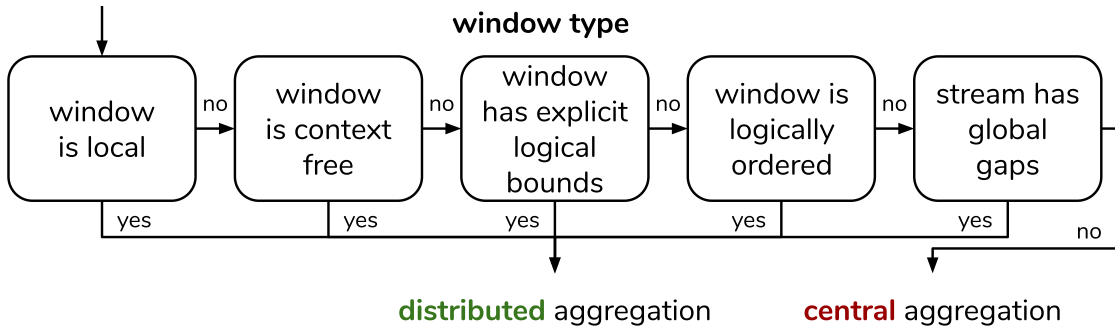


Figure 16: Decision flow chart - process window distributed or centralized?

Time, Punctuation, and Custom Measures

The use of time as a window measure is covered by all window types above and imposes no further restriction on them. Punctuation windows are covered in detail in FCF windows, and can generally not be applied to other window types. The use of custom window types, i.e. window definition by some arbitrary property of the available data, is covered to some extent in FCF and FCA windows and their global window interpretations. The use of custom windows generally makes it hard to exploit distributed aggregation and requires centralized processing in many cases. However, most queries rely on standardized window types and measures, as custom windowing logic is often error-prone and inefficiently implemented in modern SPE's due to the lack of available optimizations.

Count Measures

Using a count-based measure for distributed aggregation has similar room for interpretation as FCF and FCA windows. As soon as a global ordering is required, all events have to be collected at a central instance for correct processing. However, knowledge of the underlying data can be used to optimize count-based windows and avoid central data collection and aggregation. A count-based window can generally be interpreted in the same four ways as presented for FCF windows, namely with local windows, with explicit bounds, by processing order, or by global gaps. Interpreting local count-based windows is identical to FCF and FCA windows and has the same advantages. Processing count windows by their order is possible in the same examples as mentioned above, i.e. if there is explicit knowledge that the data supports such an assumption. Gap-based window merging is theoretically possible for count windows but requires some notion of “time” passing, which needs to be emulated with, e.g. dummy events. In general, count-based windows explicitly require global ordering, making them unsuitable for distributed aggregation.

In summary, we show that different window types can profit from distributed aggregation in certain cases and under certain assumptions. We show this decision flow in Figure 16. While CF windows can always be computed distributed, FCA windows require certain characteristics of the underlying data stream to be able to benefit from a non-centralized aggregation. FCF windows can be processed on different nodes in some cases, e.g. with explicit window punctuations or under similar assumptions as with FCA windows. Having a count-based measure requires centralized aggregation in most cases, only allowing for distributed aggregation with specific count-based interpretations of the stream.

3.3 Aggregate Functions

The type of aggregation function determines which data needs to be transferred between nodes for a distributed aggregation. In this section, discuss the two classes of aggregation functions, as presented in Section 2.5.1, *decomposable* and *non-decomposable* functions, as well as the impact of non-commutativity on distributed aggregation.

Non-Commutativity
We first discuss the commutative property ($\texttt{combine}(x, y) = \texttt{combine}(y, x)$) of an aggregation function, as this is orthogonal to the two classes of functions, but also impacts the aggregation. If an aggregate function is non-commutative, i.e. $\exists x, y : \texttt{combine}(x, y) \neq \texttt{combine}(y, x)$, we cannot pre-aggregate any data. For example, a concatenation of the event values A and B can produce AB or BA. These results are not equivalent and require global ordering. As all non-commutative functions require global ordering, we need to collect all raw events at a central node and process them there. This is independent of the class of aggregation function.

Decomposable Aggregation Functions
Decomposable aggregation functions can profit greatly from distributed aggregation. As the name implies, they can be decomposed and processed in parallel or, as in our case, distributed. In order to process decomposable functions in a distributed manner, we simply need a definition of the three methods mentioned above, `lift`, `combine`, and `lower`. Locally, all values are `lift`'ed into an intermediate representation, which by definition of decomposable aggregate functions has a constant size. The fixed-sized pre-aggregated value can then be sent between nodes and `combine`'d as needed. Eventually, the final intermediate representation is `lower`'ed and the result is emitted. This fixed-size intermediate representation

is very beneficial for distributed aggregation as the amount of data that needs to be transferred is reduced to a minimum compared to sending all raw events. In order to reduce the size of the pre-aggregate, a minimal intermediate representation should be chosen for decomposable functions.

Non-Decomposable Aggregation Functions

In contrast to decomposable aggregation functions, non-decomposable (i.e. holistic) functions require all data to calculate correct results, thus, they cannot profit as much from distributed aggregation. We again need a definition of the non-decomposable according to `lift`, `combine`, and `lower`. Compared to decomposable functions however, the `lift` and `combine` methods are pre-defined. For any holistic function, `lift` simply converts the single element x into a representation of a single-element collection containing only x. To `combine` two pre-aggregates, all non-decomposable functions must merge the intermediate collections. This step could also be used for some pre-calculation, e.g. for `median` a sorted collection is required, so the two collections could be merge-sorted to avoid this step later on in a potentially very large collection of raw events. However, this is not always needed and should only be done if it is actually beneficial, in order to reduce the cost of the `combine` method. The negative impact of holistic functions on distributed aggregation is the amount of data that needs to be transferred between nodes in the system. Compared to a fixed-size value in decomposable functions, a non-decomposable intermediate representation grows in $O(n)$ with the number of events for a given window. In Section 4.4, we discuss possible optimizations that can be made in certain scenarios, but there is no universal improvement for all holistic functions. After merging all pre-aggregate states, all raw events are finally collected on a single node, where `lower` can be called to perform the actual aggregation. This also has a negative impact on the performance compared to decomposable functions, as they can distribute the computational load across multiple nodes.

In summary, there are two main characteristics that determine whether an aggregate function can be computed distributed or not, *i)* its commutative property and *ii)* if it is decomposable or not. We show this in Figure 17. Non-commutative functions always requires a global ordering and cannot be processed distributed. Non-decomposable functions do not necessarily require a global ordering but as they need the entire data to perform the aggregation, they also cannot generally benefit from a distributed aggregation. However, there are optimizations that can be made to increase the performance of holistic functions, which we discuss in Section 4. Together with the window type decision flow in Figure 16, we can decide whether a window-aggregation combination can benefit from a distributed aggregation. The decision flow can be interpreted as a logical and, the computa-

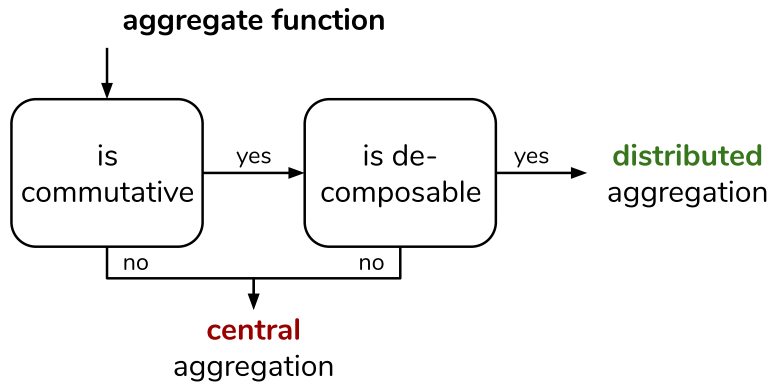


Figure 17: Decision flow chart - is aggregation distributed or centralized?

tion can be performed on multiple nodes only if both flows lead to a distributed aggregation.

4 Distributed Data Aggregation Implementation

Disco's primary goal is to pre-aggregate windows close to sensors to avoid the expensive transmission of raw events through the system. To this end, Disco runs distributed on multiple nodes in a tree-like network structure. In this tree-like structure, each node is responsible for the aggregation of all data and windows in its sub-tree, which leads to an even distribution of computational cost between nodes in the network. On the lowest level, windows are created directly based on the incoming sensor data, while higher levels in the tree are responsible for merging windows and their partial aggregates for incoming partial windows. The last level finally emits the window aggregate to the user.

In this section, we present our implementation of the theoretical window merging concepts, as described in Section 3.2, in Disco. For this, we first give a high-level architectural overview of Disco (see Section 4.1). We present our window representation in Section 4.2, followed by the different window types and how they can be merged efficiently (see Section 4.3). We then discuss how to implement the aggregation of data on multiple nodes in Section 4.4. Following this, we present the three main components of Disco in detail and show which part of the window aggregation they perform (see Section 4.5). In particular, we discuss window creation on child nodes in detail in Section 4.5, for which we use the general stream slicing library *Scotty* [58]. We then also discuss window merging on intermediate nodes and final result calculation on the root node. Finally, in Section 4.6 we conclude this section with a short summary of our proposed distributed data aggregation approach.

4.1 Disco Architecture Overview

In this section, we present a high-level overview of the interacting components in Disco. The structure of our system is shown in Figure 18. Disco's topology is structured in a tree-like fashion, where each level of the tree has a different core function. There are four main components or levels in Disco:

1. The external sensor devices that produce continuous data and send it to our system.
2. The child nodes that receive the raw sensor data, process it to create windows and partial aggregate, and send the intermediate window results up the hierarchy.

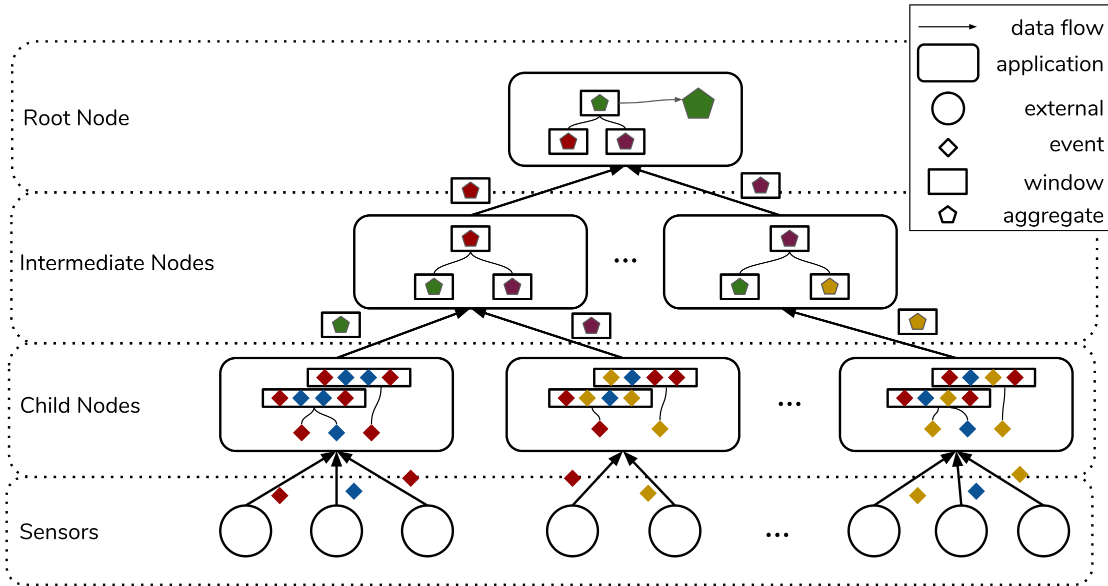


Figure 18: Disco high-level architecture overview.

3. The intermediate nodes, which collect intermediate windows from lower-level child nodes, merge the window results, and pass the updated result on.

4. The user-facing root node that receives the final intermediate windows from each sub-tree, merges them to a global window result, and finally emits the result to the user.

We now provide a bottom-up flow of events from the sensor nodes through our system to the final result at the root. First, an arbitrary number of external sensors produce data, which each sensor then sends to a chosen child node in the system. The data consists of single events with an event value and timestamp. A child node constantly receives the events from one or more sensors and aggregates them according to the requested window and aggregation semantics. If no distributed aggregation is possible, the child node can only pass on individual events. In that case, all nodes simply pass the raw events to the root, where the data is then aggregated on a single node. However, at this point we describe the part of Disco in which we can actually benefit from distributed aggregation, so for the rest of the overview, we assume that the window aggregation can be distributed across nodes. After a window is complete, the child node sends the intermediate window containing the pre-aggregate to an intermediate parent node. We note here that there is no theoretical limit to the number of child nodes that can be present in the system. It is only limited by the processing capacity of the nodes in the level above, which need to handle the intermediate results of all children.

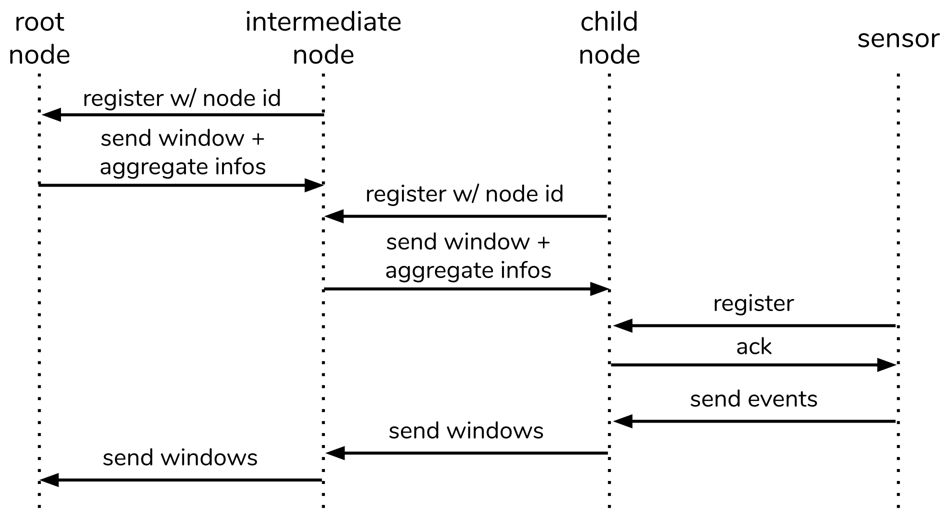


Figure 19: Inter-node communication on system startup.

Once an intermediate node receives a window from a child node, it merges it with other windows that it has received from other child nodes. The partial aggregates of the children then become a combined aggregate, which in turn can be sent to the root. We again note that there is no theoretical limit to the number of intermediate nodes that can be present. Also, there can be an arbitrary number of intermediate levels. Figure 18 shows only one level, but a Disco topology could consist of any number of levels. The intermediate nodes simply need to receive intermediate window results from a lower level, merge them correctly, and then pass them on to a higher level. The lower and higher levels do not need to be child and root nodes, respectively. In this case, intermediate nodes are connected to other intermediate nodes.

Finally, the root node receives all intermediate windows from the level below and merges them to the final aggregation result. The final result is then emitted by the system and presented to the user. Contrary to the other levels, there can be only exactly one root node, as there can be only one final aggregation result. This system is designed to run constantly as an SPE would, meaning that data is continuously ingested, windows are continuously created, intermediate results are continuously merged, and final results are continuously emitted. We present each component in detail in Section 4.5.

We now present an overview of the communication between nodes on system startup in Figure 19. The root node is the interaction point with the user. Disco runs in two phases, the initialization phase and the execution phase. In the initial-

ization phase, all nodes register at their parent node and store metadata about the requested windows and aggregations. Before the systems starts, the user defines all queries, i.e. windows and aggregate functions on it before the system starts. When the system starts, the root node waits for intermediate nodes to register with their node id. On receive, the root node then stores the individual node ids in order to know when it has received windows from all nodes during window merging. The root then responds with the requested windows and aggregate function, as provided by the user. Once an intermediate node receives the window information, it stores this to know for which windows it will receive partial aggregates. The intermediate node is then ready to receive registration messages from any direct child nodes. Analogously to the root, it stores the node id and responds with the windows and aggregate functions. The child node then knows for which windows and aggregates it needs to perform aggregations. After registering at the intermediate node, the child node waits for sensors to register. They do not need to send a sensor id, as the child node makes no distinction between events from different streams, they are seen as one logical stream. Once the child node has acknowledged the sensor registration, the sensor can begin sending data.

Now, Disco moves from the initialization phase to the execution phase. In this phase, the requested windows and aggregations are created, merged, and emitted to the user. Sensor events are continuously windowed and aggregated on the child nodes, which in turn send the windows to the next higher nodes, which again merge all incoming windows and forward their updated windows. This is done until all merged windows reach the root, which then emits the final result. We note here that our current implementation does not support the dynamic entering and leaving of nodes. The number of direct child nodes need to be known in advance, so each node can wait until all nodes have registered. Studying a dynamic node setup poses an interesting challenge for future work, as each node needs to keep track of which nodes will still send windows, from which ones it has already received windows, and which merging step includes how many expected windows.

4.2 Window Representation

In order to aggregate windows in a distributed manner, we need to establish a common format in which we can represent and transfer windows between nodes in the system. The window representation is shown in Figure 20. Based on the example query "tumbling window with a length of one second and the aggregation function `sum`", we describe the individual fields.

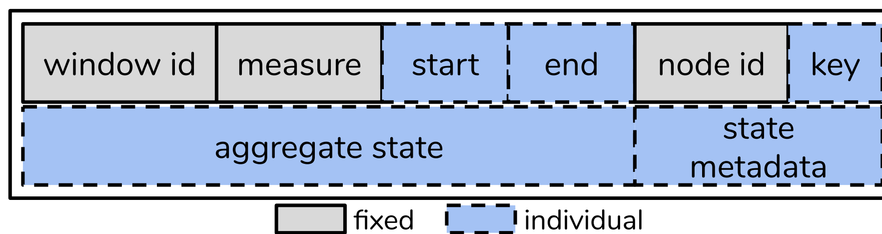


Figure 20: Window representation.

window id: The window id uniquely identifies the requested user-window. In our example, the tumbling window would receive the id 1. If a user also requests a sliding window, this would result in the window id 2. The id is unchanged throughout the system's runtime and is used to map from the internal window representation to the user's result window.

measure: The measure specifies whether the window uses time or tuple-counts as its bounds. This also stays unchanged throughout processing. In our example, the measure is *time*.

start: The start field stores the start-bound value and is different for each window instance. For our tumbling window, the start would be at second 0 for the first instance, at second 1 for the next instance, etc.

end: The end field is analog to the start field but it stores the end-bound, i.e. second 1 for the first window, second 2 for the second, and so on.

node id: The node id field is used to identify at which node the window instance was created. This tells us from which children we have received a window for a certain period so that we know when we have received all windows and have a complete result. This field stays the same for each window, as the id of each node does not change.

key: The key field stores the key for which this window was created. If a stream has, e.g. five unique keys A, B, C, D, and E, in our example query for each one-second interval five windows will be created, one for each key.

aggregate state: A window is also associated with some partial or final aggregate state, which has an arbitrary size compared to the fixed-sized header fields. In our example query, the state associated with each window would be a single integer value representing the current sum. For more complex aggregations, this state could also be, e.g. a list or a tuple.

metadata: The metadata field is used to store some information about the state, e.g. which aggregation function is used and to which class of aggregate function it belongs.

The combination of window id, start, end, node id, and key uniquely identifies a window when transferring it between nodes. For a requested window, there are many instances so we need the start and end bounds. For context free windows, the start and end bounds are equal on all nodes so we need the node id to distinguish between them. Finally, for any window we can have multiple keys, so we need to store the key with each window for the final result. This window representation can easily be serialized and transferred between nodes with a simple custom string-based protocol.

4.3 Window Types

Depending on the window type, we apply different merging strategies. To this end, we introduce a sub-component of our system, the *Window-Merger*. The Window-Merger is responsible for collecting incoming windows, merging them if possible, and emitting the merged windows to the next higher level. The decision on whether a distributed aggregation is possible or not, as shown in Figure 16 and Figure 17, is performed by the Window-Merger during system initialization and then executed by it. For each incoming window, the Window-Merger then decides if a merge is possible and needed. Should this be the case, it compares the window with previously collected windows and merges accordingly. In case of a merge, once it has received all pre-aggregates which belong to that window, the updated window is passed one level higher. The Window-Merger is responsible for all nodes in the sub-tree below it, receiving windows from its direct children, which in turn have possibly already merged other pre-aggregates belonging to the final window result. We now discuss the exact merging strategy for each window type *context free*, *forward-context free*, and *forward-context aware*, as well as for count-based windows.

Context Free Windows
For CF windows, the Window-Merger stores all windows in one map, where the key of each entry is the window id and the tuple of ($window\ start, window\ end$). We show this in Figure 21. When the Window-Merger receives a window, it contains the pre-aggregates for each key in the stream. If the window did not contain all keys, we would need an extra control message indicating whether a node has sent all data for a given window or not. As the window is triggered for all keys at the same time, there is no additional waiting before sending them together and we then automatically know that we have received all data from that node.

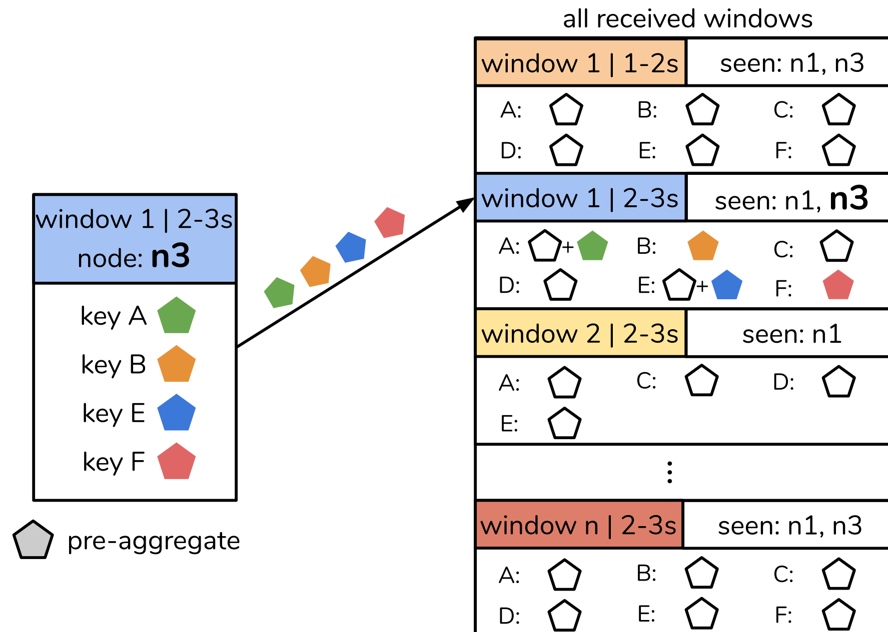


Figure 21: Context Free Window-Merger.

In Figure 21, the Window-Merger receives a window with the id 1 for the time period of 2 to 3 seconds from node $n3$. It then looks up if it has received pre-aggregates for this window from other nodes and then merges the aggregates if needed. Keys A and E already have some state and are merged with the incoming values. The keys B and F have not been seen before, so their state is simply set for the given key. Keys that are not present in the received window are left untouched. Once the states have been updated, the Window-Merger adds the node id $n3$ to the list of seen nodes. Once that list is complete, i.e. all nodes have sent data for a given window, the window and all key states are passed on to the parent node. The window id, the start bound, and the end bound are all left unchanged before passing on the window. However, the node id must be set to the node on which the Window-Merger is located. As each level only know its direct children, we need to set the node id to the current node so that the next Window-Merger can track the completeness of received windows in regard to its children.

On the next higher levels, the exact same steps are then performed with different node ids until the window eventually reaches the root and the final result is emitted. There is not necessarily an order in the state of all windows in the Window-Merger as it depends on the order in which the nodes send their data. In this example, node $n2$ has not sent their data for window 1 from 1 to 2 seconds, so even though later windows have arrived, it is not complete.

Context Aware Windows
For the Window-Merger, there is no distinction between FCF and FCA windows, so we group them into *context aware* windows, as they both require some kind of context. We now discuss the different interpretations for context aware windows, namely local windows, merging by explicit bounds, merging by processing order, and merging by global gaps.

For local windows, there is no need for a Window-Manager because it can simply pass on the received window. There is no other window containing relevant data, so we do not need to store or merge it. Also, there is no need to adapt the node id on each level, as it has no real meaning without the merging step.

Processing windows with explicit bounds, e.g. through equal punctuations ingested by the system into all stream, can be done similarly to the merging of context free windows. The map containing the received windows is the same, as we expect each window to come with explicit and equal bounds. Compared to CF windows, however, these aren't necessarily time bounds, i.e. seconds or minutes, but can take on any form of logical bound, e.g. a counter or certain keywords. As long as the Window-Merger can identify matching bounds, we can use the same approach.

Merging windows in the order they are processed in can also use this map-based window state. In this case, the key for each window in the map is the window id and a counter, as opposed to start and end bounds. For this, we need an extra small map that counts the number of received windows for each node. For an incoming window with the id 2 from node $n1$, the Window-Merger looks up how many individual windows with the id 2 it has received from node $n1$. It then assigns that window the retrieved count value and increases the count in the map. The window can then be looked up in the window state map using the window id and its assigned count value. The same completeness condition as for context free windows applies here too.

Merging windows by global gaps in all streams requires a different approach than the above map-based window state. We present this approach with the example of session windows because that is the only way to process them. The global gap based approach for non-session windows can be viewed completely analogously by simply setting the gap used in session windows to 1 in the smallest available unit for the given window measure, as described in Section 3.2. For session windows, the gap is defined by the window itself.

The merging of session windows is shown in Figure 22. The Window-Merger receives a session window for the window function with id 1, which has a defined

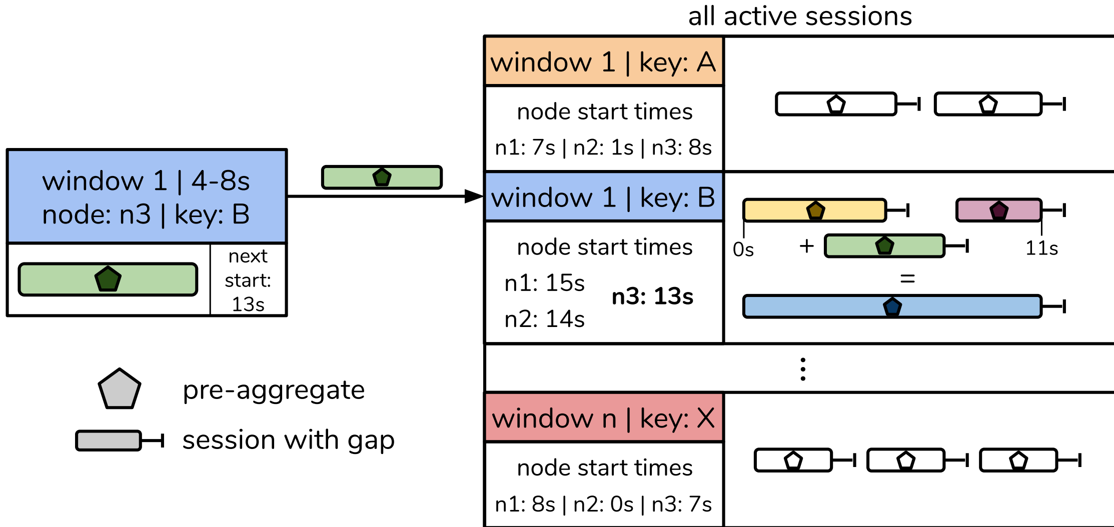


Figure 22: Session Window-Merger.

session gap of one second. The window for key B from node $n3$ starts at four seconds and ends at eight seconds. It can then look up all previously received sessions for this window and key and check if there is any overlap between them. In our example, the incoming session overlaps with two sessions from other nodes, combining them into one large session. We also see here that an overlap does not necessarily need to be based on start and end bounds but can also include the session gap. As mentioned above, in order to know that there is no more data for a given session s we need to collect windows from all nodes that start after the end (including the gap) of s. This can lead to a very high latency for session windows, as we need to wait an entire extra session length in order to know that the current session is complete.

To avoid this high latency, we explicitly send the next session start timestamp with the window. We always know the next start, as it caused the previous session to end, so we can easily transmit this information with the window. As the explicit next session start is only available when a session is created, we do not have this on nodes where sessions are only merged. However, we still want to pass this information up the hierarchy in order to avoid waiting for all completed windows after a given session. Once a session is complete in the Window-Merger, it looks at the session starts that it received from its direct children and computes the earliest start from them. At this point, the Window-Merger has not necessarily received any data for a new session but it can guarantee that there will be none before the session start, as all children have guaranteed it themselves. We then know that there is no more data between the current session end and the next session start.

This information is then passed on to a higher Window-Merger where it can be processed identically to the explicit session start from a created session. Thus, from the Window-Merger's perspective, it is irrelevant how the next session's start was determined, as long as it is correct.

In our example, the next session for window 1 and key B on node $n3$ starts at 13 seconds. Once the Window-Merger has successfully merged the overlapping windows, it updates the next session start for the node it has just received the window from. We now have a merged window from zero seconds to eleven seconds and all nodes' next active sessions start after this session's end (including the gap). The Window-Merger now knows that there will be no more data for this session so it can be sent to the next higher level for further merging or the final result can be emitted. We also see that for window 1 and key A, the first session window is not complete because node $n2$ has an active session starting a one second, which will overlap with the previously collected windows.

Count-based and other Non-Distributable Windows
As we generally assume that count-based windows cannot be aggregated in a distributed manner, the raw data needs to be forwarded to a central node for processing. This is also the case for FCF and FCA windows that cannot be interpreted in a distributable manner. In that case, there is no Window-Merger, as there is only one final window and no intermediate ones that need merging.

4.4 Aggregate Functions

So far, we have discussed the different strategies for merging windows depending on their type and interpretation. In this section, we now examine how exactly the window state is combined, depending on the class of aggregation function. We introduce another sub-component of Disco, the *Aggregate-Merger*. The Aggregate-Merger is an overarching component responsible for parsing the received windows and determining which specific Window-Merger is required for the current window, based on the window's metadata. So far, there has been only one generic Window-Merger but we need to distinguish between a decomposable Window-Merger and a holistic Window-Merger for the different aggregation functions. The window merging logic on a high level is identical between the two, but the aggregate merging is very different. We now present the aggregate state merging for decomposable and non-decomposable functions.

Decomposable Functions
The decomposable Window-Merger is used for all distributive and algebraic aggregate functions. Its main task, apart from window merging, is to simply call

`combine` on two pre-aggregates. All logic for this is implemented in the user-defined aggregate function and the decomposable Window-Merger does not need to know about it. Once a window is complete, the decomposable Window-Merger simply calls `lower` on the intermediate representation to retrieve the final value in the correct output type. In order to transfer partial aggregates between nodes, functions in which the intermediate representation is not a basic numeric value, e.g. `avg`, need to specify an intermediate string-based representation for the aggregate. Internally, we add a small wrapper around the original aggregate function, which is then responsible for serializing and de-serializing from and to that format.

Non-Decomposable Functions
Handling non-decomposable aggregate functions requires different steps than the decomposable Window-Merger performs. To understand these steps, we need to look at our window creation process. For all previous window merging concepts, the exact windowing technique or algorithm was not relevant. The transferred windows simply contains a pre-aggregate that can be merged, regardless of how that pre-aggregate was calculated on the child node. For holistic windows however, we make heavy use *slices*, as presented in Section 2.5.3.

A slice is a bounded chunk of streamed data that does not overlap with other slices, i.e. each event belongs to exactly one slice. This non-overlapping property can be utilized to transfer each raw event exactly once in a logical group of events, which reduces the network cost because we do not need to send a TCP-packet per event. We discuss the exact slice creation and the benefits thereof in Section 2.6 and show how we can efficiently transmit them in Section 4.5. The main idea behind using slices for holistic aggregation is that once a slice has been sent it is immutable and does not need to be updated, nor do single events from it need to be sent again.

To work with slices instead of pre-aggregates, we use a holistic Window-Merger. The window merging logic for the holistic Window-Merger is identical to that of the decomposable Window-Merger. However, we do not simply `combine` the partial aggregates. If the Window-Merger is not responsible for the final aggregation result, it simply keeps track of which slices have come through and passes them on to the higher level. By keeping track of the node id and slice start, we avoid duplicate sending as we know exactly which slices have been sent already and we do not need to store the slice content itself. The Window-Merger responsible for producing the final result then looks at the slices that belong to the complete window, based on their bounds, and adds them together. The holistic function then has the entire set of events to perform the correct aggregation.

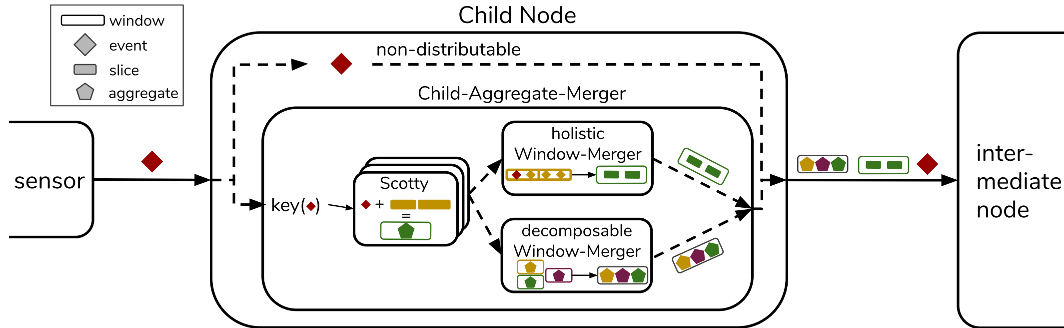


Figure 23: Disco's child node component.

4.5 Disco Components

We now present the three main components of Disco, the child node, the intermediate node, and the root node, to show which Window-Merger tasks they perform for which windows and aggregations.

Child Node

The child node is the first component of Disco that processes data. A child node communicates directly with the sensors and receives their raw events. From them, it creates the requested windows that are then passed on to intermediate nodes. To create windows and their respective slices, we use *Scotty* [58]. Scotty is an open-source windowing library for arbitrary windows on the basis of slices. We describe the exact slice and window creation in detail in Section 2.6. We show the flow of events and the integration of Scotty in child nodes in Figure 23.

When a child node receives an event, it retrieves the event key and processes the event with the Scotty instance for that given key. Scotty determines which slice the event belongs to and `combines` the slice state, i.e. the pre-aggregate, with the incoming event. At regular intervals, we check whether windows are complete. After a specified maximum lateness for events, we then trigger the complete windows for all keys. An advantage of window creation close to the data source is the reduced uncertainty of the event transmission duration. If an event is simply forwarded from node to node it incurs additional latency at each network hop. This can lead to the necessity of a very long maximum allowed lateness at the central processing node, as it essentially needs to wait for the slowest sub-tree to send the complete results. In many cases, however, there is no need for this long waiting period because the data flows sufficiently fast. Having the window creation directly at the data sources allows for a much more fine-grained lateness dependent on the individual sensors. Even if a sub-tree is slow and the window

reaches the root very late, the root can explicitly trigger the complete windows once it has received all sub-windows. It does not need to wait implicitly until all data has, or has not, arrived.

Depending on the type of aggregate function for a window, the child node either processes the window in the holistic or decomposable Window-Merger. For decomposable functions, the Window-Merger needs to merge the keyed windows into the form described above, where the pre-aggregates for all keys are present in one window message. This is logically equivalent to merging regular windows just without combining the window states but rather each one. For holistic functions, the child node uses the holistic Window-Merger. Here, we handle the individual slices instead of pre-aggregates. We do this to avoid duplicate sending of events. Each slice contains its unique raw events and a slice can belong to multiple windows. We do not want to send the same slice for different windows, so we keep track of the previously sent slices. We do this, as described above, in a simple set containing triples of the node id, the key, and the slice start. This triple uniquely identifies each slice, as two slices on the same node for one key can never have the same start bound or else they would be overlapping We can then discard a slice if it has already been sent. This can lead to windows being sent without slices, but as they are already present on the node above, that does not influence the correct computation.

For non-distributable windows, the child node simply forwards each raw event as the need to be collected at the root node for final processing. The two streams in the figure above are not mutually exclusive, i.e. if a count-based window and a tumbling window are requested, we forward the event and process it regularly. Thus, a child node can send three different messages to the intermediate node one level higher in the network, *i)* a single event, *ii)* a window with pre-aggregates, and *iii)* a window with slices.

Intermediate Node

We show the flow of incoming data for intermediate nodes in Figure 24. An intermediate node can receive single events and windows. Raw events are needed for non-distributable windows with a global ordering. For these, the intermediate node does nothing but forward them to the next node. On intermediate nodes, windows are merged based on their key, so we use a window id and key to uniquely identify the individual window. When an intermediate node receives a window, it parses the window, retrieves its key, and triggers the responsible Window-Merger with the extracted information. For decomposable functions, the intermediate node calls the decomposable Window-Merger, which in turn merges windows if needed and combines their state into an updated pre-aggregate. That pre-aggregate is then

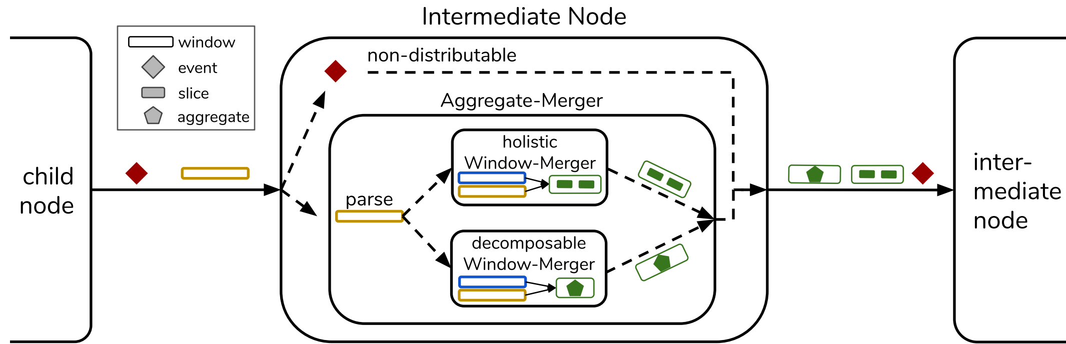


Figure 24: Disco's intermediate node component.

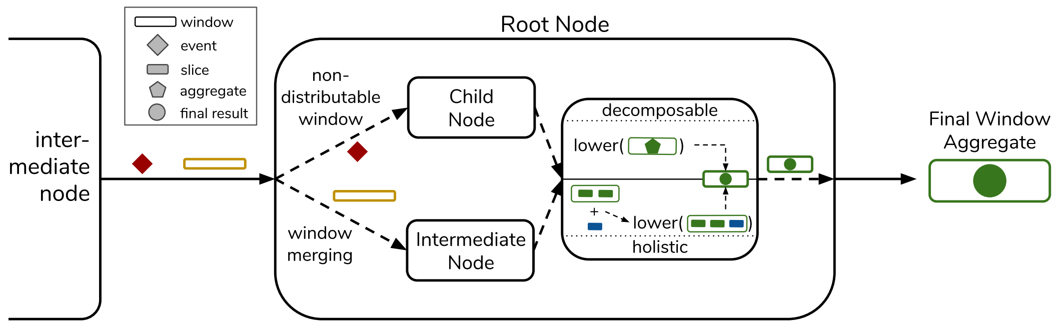


Figure 25: Disco's root node component.

passed on to the next node in the hierarchy together with its updated window. For holistic functions, the intermediate node calls the holistic Window-Merger, which merges windows if needed, computes which slices need to be sent with the window, and forwards those. For this, it uses the same mechanism as the child node by tracking the triple of node id, key, and slice start to avoid duplicate slice sending. Again, an intermediate node can send the same three message types to the next node in the hierarchy. As the message formats are identical, it is not relevant for an intermediate node whether it receives messages from a child node or another intermediate node. This allows us to scale to an arbitrary number of intermediate levels.

Root Node

The root node is the final node in our distributed aggregation hierarchy. We show the data flow for a root node in Figure 25. The root node is essentially just a special child and intermediate node combined. It can receive messages from both child and intermediate nodes as their message formats are identical. When it receives a single event, we know it is for a non-distributable window. In that case, the root node acts as a child node. The event is parsed and the key is extracted.

Based on that key, the corresponding Scotty instance is called to process the event. We employ the same window completion check as for child nodes, i.e. after a certain lateness period, but we do not collect all window states into a single window but process each complete window individually and `lower` the aggregate in the window state. We distinguish between decomposable functions, for which the aggregation is already complete because of the Window-Merger, and holistic functions, for which we have a store of all slices that need to be assigned to the final window. For decomposable functions, we can simply call `lower` on the final pre-aggregate and emit the final window result. For holistic functions, we need to get the corresponding slices for the final window, merge their events, and then call `lower` on all events for that window. Again, we can then emit the final result. Holistic functions impose a high computational cost on the root node, as it is the only node that can perform the final aggregation. However, for decomposable functions, most of the computation has been performed on other nodes and the root node has to merge only a few windows.

Concluding the implementation details, we give a short overview of some code-related specifications of Disco. Disco is an open-source[1] project implemented in Java 11. All node inter-node communication is done via JeroMQ[2], a Java implementation for ZeroMQ[3]. We use JeroMQ version 0.5.0, which supports ZeroMQ Message Transport Protocol 3.0[4]. So any external program that supports this standard can communicate with our system. We also use a forked version of the open-source library Scotty[5] to add a few additional fields to windows and to make minor changes to existing classes. This fork is also available open source[6].

4.6 Summary

In this section we presented Disco, a stream processing framework for distributed window aggregation in Fog computing networks. For this purpose, we looked at the theoretical concepts behind distributed aggregation for arbitrary windows. We showed that context free, forward-context free, and forward-context aware windows can all profit from distributed aggregation in some cases. We also discussed the impact of the different classes of aggregation functions. Decomposable functions can benefit immensely from a distributed aggregation setting, whereas

[1] `https://github.com/lawben/distributed-scotty`
[2] `https://github.com/zeromq/jeromq`
[3] `https://github.com/zeromq`
[4] `https://rfc.zeromq.org/spec:23/ZMTP`
[5] `https://github.com/TU-Berlin-DIMA/scotty-window-processor`
[6] `https://github.com/lawben/scotty-window-processor`

holistic functions can profit only marginally by sending data in logical chunks and thus reducing network cost. We showed how these theoretical concepts can be implemented in a multi-node tree-like network structure by designing three core components around them. While child nodes receive all the raw data, in most cases we can avoid sending raw tuples between all other nodes in the system through early window creation and aggregation.

5 Evaluation

In this section, we present our evaluation of Disco. First, we discuss our methodology and environment set up in Section 5.1. We then present detailed benchmarks of Disco compared to a centralized data aggregation implementation in Section 5.2. The centralized implementation represents current state-of-the-art aggregation concepts that are not optimized for Fog-like network structures. In our evaluation, we use the three main metrics: throughput, latency, and network cost. Finally in Section 5.3, we conclude and discuss the main findings of our evaluation.

5.1 Setup and Methodology

Before we evaluate Disco, we first present our general benchmarking approach. To this end, we briefly discuss how we generate test data and how we run our distributed benchmarks.

Data Generation
For the generation of test data, we follow Karimov et al. [36], which allows us to generate events at a specific rate, e.g. one million events per second, in order to simulate a sensor. We use a separate data generator process that sends events directly to our nodes. The generator is always exclusively located on a node, to avoid resource contention while benchmarking. We do not use message brokers such as Apache Kafka [38] as they can quickly become a bottleneck while testing the system [36].

We always evaluate the event-time latency of Disco, i.e. the time from the event creation to the emission of the final window result containing the event and not processing-time latency, which describes the time it takes to process a single event from entering the system to exiting it. For this purpose, we use two threads in our data generator, a producer and a sender thread. The threads communicate via a shared queue. We do this to avoid coordinated omission [56], i.e. the discrepancy between event-time and processing-time latency. If we do not generate events independently of sending them, we cannot guarantee generation at the given fixed rate. Friedrich et al. show that coordinated omission can lead to a significant underestimation of latency [20].

The producer thread divides the requested rate into fixed-size chunks and then generates the data according to these chunks with a sleep period in-between. After

a chunk is complete, the producer calculates how much time is remaining in the current generation-second and sleeps for a portion of it, depending on the number of remaining chunks. We do this to ensure an even distribution of event times over the one-second period. Before benchmarking, we ensure that our generator can produce data faster than the system can handle the incoming events. Otherwise, we would simply measure the performance of our event generation. The sufficient speed of our generator can also be seen below in Figure 33.

In general, we measure the event-time latency for a given window as the difference between the event time of the last event in the window and the time the window result was received [36]. The highest timestamp in the window is the earliest time at which the system could theoretically know that the window is complete. The generator creates an event at a given timestamp, the system processes it in a window, and finally, the root node emits the result, so we measure the time between creation and the root node. As these processes are running on different nodes, we synchronize all nodes' clocks via NTP to ensure correct measurements.

Another important concept is sustainable throughput. We define a throughput as sustainable if the system can handle the incoming events without an ever-increasing backlog. We again use the queue between the producer and the sender to check this condition. Should the queue reach a certain size and increase monotonically from then onward for a certain period of time, we declare the throughput unsustainable. All evaluations concerning the throughput of Disco show the maximum sustainable throughput that can be achieved.

Benchmark Setup

All of our experiments run on a cluster with up to 20 nodes, depending on the required setup. Each node has an AMD Opteron 6128 @ 2.0 GHz with 16 physical cores and 32 GB of RAM. All nodes are running Ubuntu 18.04.3 LTS and are connected via Gigabit LAN. We run our code with OpenJDK 12.0.2 64 bit. We never run two or more processes on the same node to avoid falsified results because of resource contention or similar effects; thus, most of the CPUs and RAM are unused. Unless stated otherwise, each benchmark is executed for 120 seconds. We discard the first thirty seconds of each run to allow for the JVM to warm-up. A quick evaluation showed that thirty seconds are sufficient for this. We also discard the last thirty seconds to avoid problems regarding the shutdown of Disco. The middle 60 seconds represent an open-world scenario in which the system has been running for a while and does not terminate.

5.2 System Benchmarks

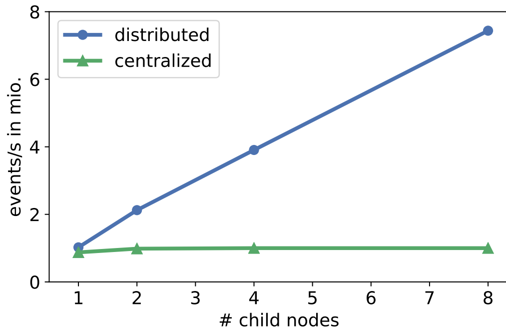
We now discuss individual benchmarks showing the performance of Disco for various metrics in different scenarios. We first study the scalability of Disco in Section 5.2.1, followed by the amount of network traffic produced in the system (see Section 5.2.2). We then evaluate the impact of concurrent windows in Section 5.2.3, the impact of many unique keys in Section 5.2.4, and the latency of different window types in Section 5.2.5. Finally, we discuss the individual performance of the child and root nodes in Section 5.2.6 and Section 5.2.7, respectively.

5.2.1 Scalability

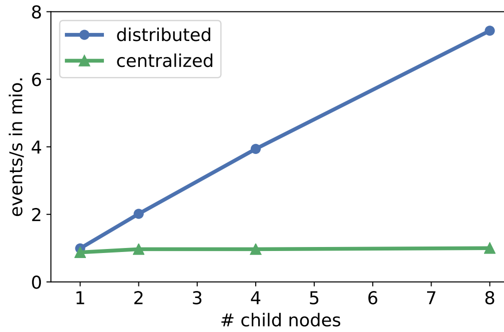
In this experiment, we discuss at the scalability of Disco. First, we measure the sustainable throughput of our system with an increasing number of child nodes for a simple tumbling window of one second. We compare the throughput of our distributed data aggregation to a centralized approach, in which each child node simply forwards the raw events to the root node for central processing. We show this in Figure 26.

In Figure 26a, we study the maximal sustainable throughput for the `max` aggregate function, which represents the group of distributive functions. Overall, we see that Disco scales nearly linearly with the number of child nodes. Already for just one child node, Disco reaches up to 1 million events per second and outperforms the centralized approach, as we do not need to send the entire raw data twice. For two child nodes, the throughput of the centralized approach does not improve, as the bottleneck is the root node that has to do all the central processing. Thus the centralized approach is limited to around one million events per second. On the other hand, Disco can now process twice as many events with approximately two million events per second. Each individual child node can process one million events and as they do not depend on each other they can both perform at this scale. The root node simply receives one partial window per second from each child node and is running at nearly no load at all. This trend continuous for four and eight nodes. Disco processes up to 4 million events per second for four nodes and 7.5 million with eight nodes. For eight nodes we do not achieve an eight-fold increase any more because the network of our cluster is reaching its limit.

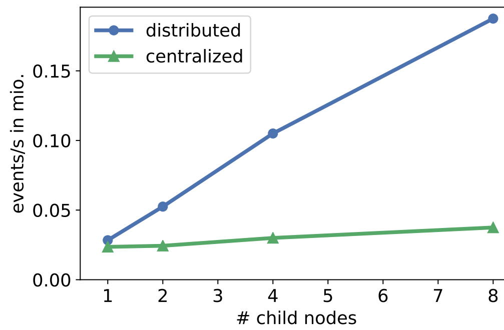
In Figure 26b, we study the throughput of the `avg` function, representing algebraic aggregations. We observe the exact same behavior as before but with a minimally lower throughput of a few thousand events per second. This is due to the fact that



(a) `max` aggregate function



(b) `avg` aggregate function



(c) `median` aggregate function

Figure 26: Scalabilty of Disco for all three classes of aggregation functions

the calculation of the average requires a bit more computation than a simple `max` comparison.

In Figure 26c, we evaluate the `median` as a representative holistic function. Compared to the previous results, the median performs significantly worse than decomposable functions because the computation of the median is more expensive than for the other functions. We discuss the exact performance characteristics in Section 5.2.6 together with the child node's performance. However, the holistic median function still scales nearly linearly with the number of nodes up to eight nodes. Again, the collection of data and merging of holistic state can be performed on multiple nodes in parallel, thus allowing for this scale. As the root node receives only up to eight windows per second, the actual final median calculation is not a bottleneck.

We see that we can profit from a distributed aggregation for all classes of aggregation functions. Even for functions that require central aggregation but can be

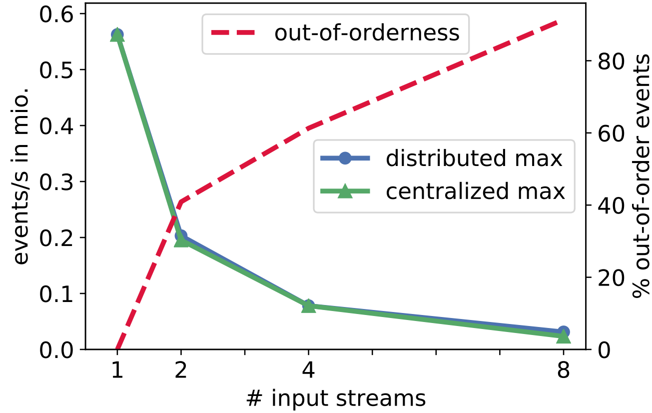
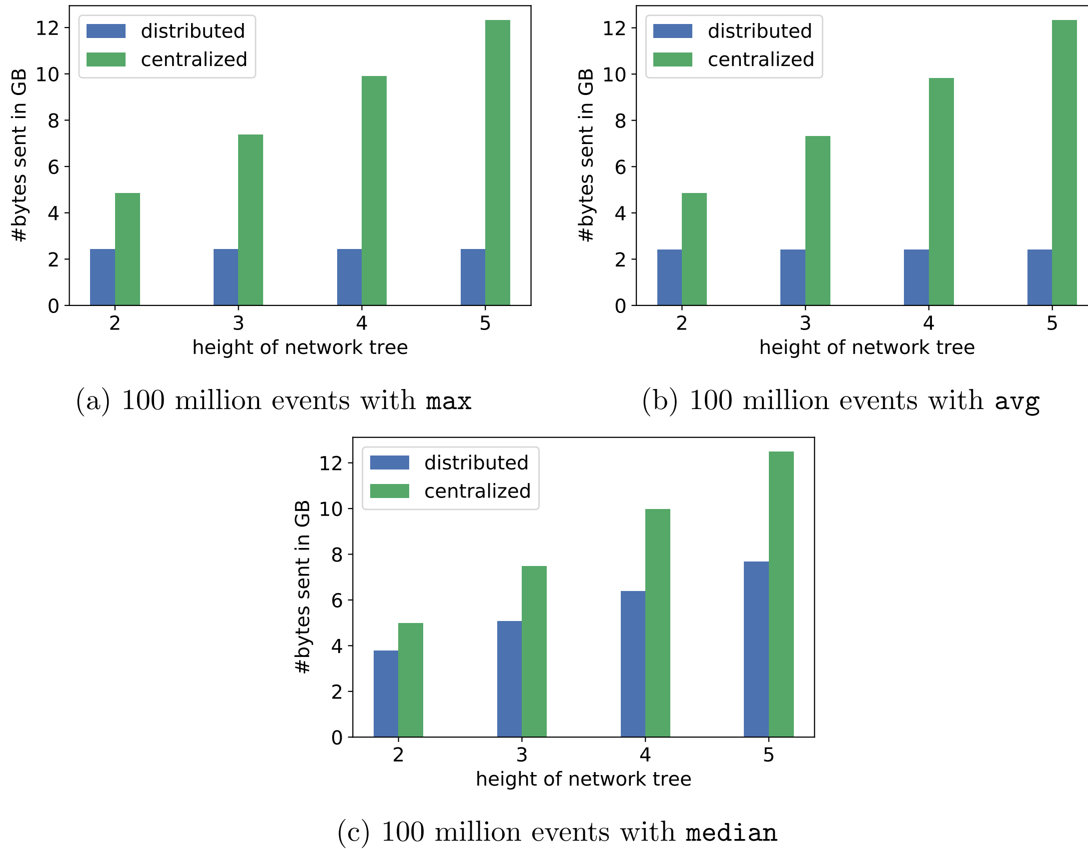


Figure 27: Tumbling count window for increasing number of streams.

windowed independently, Disco scales linearly with the number of nodes. The scale in our evaluation is only limited by the number of available nodes and the network bandwidth.

In Figure 27 we show the throughput of a count-based tumbling window with a count of 10'000 for an increasing number of input streams. We see that compared to the scale in Figure 26, the throughput actually decreases with the number of streams. This is explained by the increasing out-of-orderness of the global stream. As count-based windows require a global ordering, we need to ensure this for each incoming event. However, ensuring this order is expensive as we need to shift the individual events between the correct slice each time we receive an out-of-order event. Receiving events from multiple independent streams will lead to a higher percentage of out-of-order tuples and thus to a higher number of shift operators, which degrades the node's performance drastically. We can see that the throughput behaves exactly inverse to the out-of-orderness. For eight nodes, more than 90% of all events are out-of-order and require some kind of shifting between slices. This poses an interesting challenge for future work in which a global ordering becomes less expensive in scenarios where out-of-order events are the norm and not an exception.

Also, a count-based window represents the group of windows that cannot be processed distributed and need to be collected centrally. We see here that Disco performs on a par with the centralized approach. We can thus conclude that we benefit from a distributed aggregation in many cases and are at least as good for centralized aggregations.



(a) 100 million events with `max`

(b) 100 million events with `avg`

(c) 100 million events with `median`

Figure 28: Network cost for distributed and centralized aggregation.

5.2.2 Network Cost

We now discuss the impact that distributed and centralized aggregations have on the network. For this purpose, we measure the number of bytes that were sent through the system. In our evaluation, we generate and send one hundred million events. We evaluate this on a tumbling window with a length of one second. We create a chain of nodes to show the effect of the increasing height of the network tree, i.e. the effect of more intermediate levels.

In Figure 28a we see that the distributed aggregation in Disco has a much lower footprint on the network compared to the centralized solution. This is easily explainable by the number of raw events that need to be sent between nodes. For our distributed aggregation, all raw events are sent exactly once from a sensor to a child node. After that, our nodes send only one pre-aggregate and some window metadata once per second. As this is significantly less data than passing on the raw

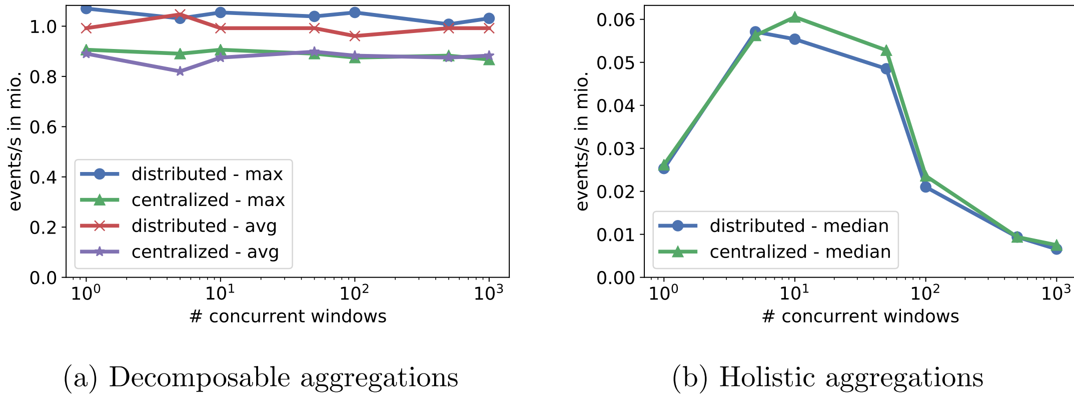
events, the network impact of Disco stays nearly constant. In our evaluation, for a tree of height two, i.e. one root node and a child node with no intermediate nodes, we send 2.416 GB of data. In a tree of height five, i.e. with three intermediate levels, we send only 2.419 GB of data. The increase for each extra intermediate level is approximately only 1 MB for one hundred million events. A centralized data aggregation, on the other hand, sends a lot more data through the system. In the tree of height two, it already sends twice as much data, as the child node simply forwards all raw events to the root. This scales with the number of levels in the tree. For a tree of height three, more than 7 GB of data are sent, for four levels 9.5 GB are sent, and for five levels we need to send 12 GB. The exact same behavior is seen in for `avg` in Figure 28b.

For holistic aggregations, Disco also needs to send all raw events to the root node. We see that for a tree of height two, we already send fifty percent more data than in a decomposable function. However, due to the sending of slices compared to individual events, we can avoid additional TCP overhead compared to sending single events. These savings become clearer for higher trees. While all single events are sent at each level for a centralized approach, slices become the smallest message unit in the distributed approach. In this scenario, we can save 50% per level compared to the centralized approach, which has a large effect once the tree becomes significantly deep. For five levels, we send only 7.5 GB of distributed slice data compared to 12.4 GB centralized single events. We note here that for certain data domains further optimizations could be made, e.g. run-length encoding on the slice data. If the data stream is the temperature reading from a thermometer, the values will rarely change. In this example, we can simply store one value and how often it occurs before changing. This could drastically reduce the size of slices if the data supports it.

Concluding, we see that sending raw events is the dominating factor in the network cost. In our distributed approach, we can avoid sending individual events between nodes for all aggregation functions classes. For holistic functions, we send events in slices which reduces the individual TCP overhead and for decomposable functions we can avoid sending single events altogether, which drastically reduces the network load.

5.2.3 Concurrent Windows

In this section, we evaluate the impact of concurrent windows on distributed and centralized data aggregation throughput. We show our results in Figure 29. For this evaluation, we test one, five, ten, fifty, one hundred, and on thousand con-



(a) Decomposable aggregations

(b) Holistic aggregations

Figure 29: Throughput for increasing number of concurrent windows.

current tumbling windows, each with a random window length between one and twenty seconds. For decomposable aggregation functions, we see in Figure 29a that the number of concurrent windows does not have an impact on the throughput of Disco. As shown in Figure 26, for a distributed aggregation we can process about one million events per second and for a central aggregation approximately 900'000. This stays the same regardless of the number of concurrent windows. Due to the stream slicing approach events are only processed once and this does not increase for concurrent windows, which leads to a constant throughput.

A different development is observable in Figure 29b for holistic aggregation functions. Here, the throughput actually increases with the number of concurrent windows. This is explained by the slicing approach. With more randomly overlapping windows, the average slice becomes smaller because there are more window starts and ends. As the holistic aggregate state is essentially a list of all events in that slice, copying becomes less expensive with smaller lists. Thus, the average event processing time decreases for smaller slices. However, this only holds up to approximately 50 concurrent windows. After that, the cost of calculating the median for each window becomes the bottleneck. For each window, all slices are collected, copied into a large list, and finally sorted to pick the middle element. Investigating optimizations for this case poses an interesting challenge for future work.

We show that the number of concurrent windows does not have a negative performance in a distributed aggregation compared to a centralized one. For decomposable functions, the number of concurrent windows has no effect on the system's throughput. Holistic functions benefit from smaller slices but only up a certain number of concurrent windows, before the slice collection and actual median calculation become the bottleneck.

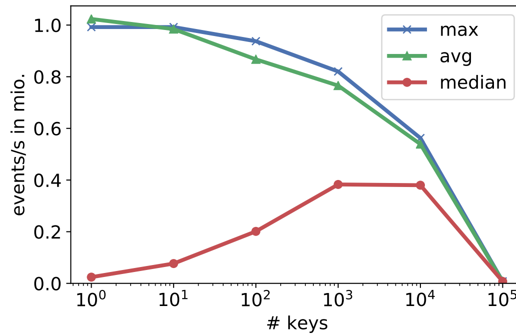
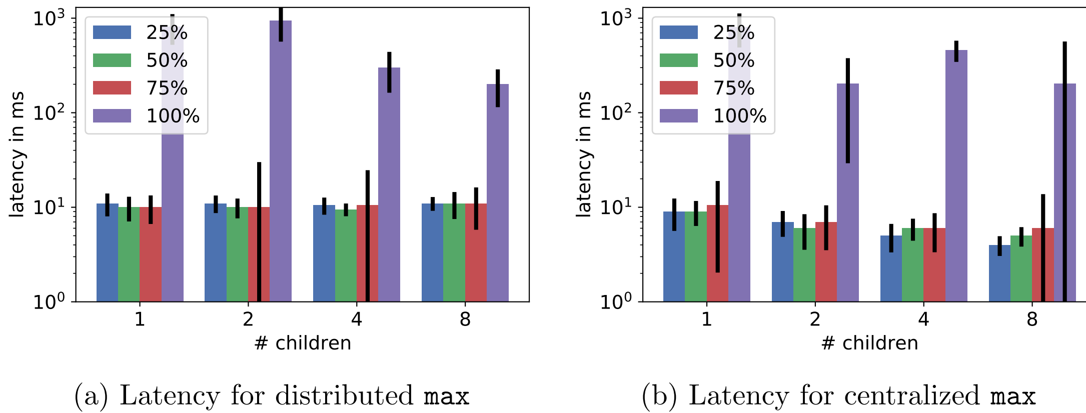


Figure 30: Number of unique keys in all streams.

5.2.4 Concurrent Keys

In this section, we discuss the impact of multiple keys per stream. We show this in Figure 30. We see that for both decomposable functions the throughput decreases with the number of keys. We see this decrease because of the extra work we require at a window's end. We need to go through all keys' slices and compute the partial aggregate for each key. Also, we send one aggregate per key, so for 10'000 keys, we send 10'000 aggregates per window, which has a large impact on the serialization and transmission time. For up to 10'000 keys, Disco still has an acceptable performance with an overall throughput decrease of 2x. However, for 100'000 keys the performance drops to merely 7000 events per second. For the `median` we see a similar effect as above, where the throughput increases with the number of concurrent windows. This is again due to the fact that the state for each key becomes smaller and less data needs to be merged and copied between slices and nodes. For 10'000 keys, the performance of `median` is close to that of the decomposable functions. However, after that, it also drops rapidly as the iteration over all keys becomes the dominant factor.

Concluding, we can see that Disco can handle up to a thousand keys per stream with a performance loss of only about 20% for decomposable aggregations and we even see an increase in performance for holistic ones. When the number of unique keys grows too large, the system needs to spend too much time on checking the states of each individual key and, is thus, heavily degrading the overall throughput.



(a) Latency for distributed `max` (b) Latency for centralized `max`

Figure 31: Latency for `max` with increasing number of child nodes.

5.2.5 Window Latency

We now evaluate the latency of Disco compared to a centralized approach in Figure 31. We again use a tumbling window with a length of one second and the `max` aggregate function. We take the previously measured sustainable throughput as 100% load, as well as a quarter, half, and three-quarters of that load. We see that for our distributed, the mean latency up to 75% load is below 10 milliseconds. For some 75% runs, the standard deviation is higher than for the lower percentages, indicating that this is a load factor at which the system's performance slowly starts to degrade. We note here that the error bars reach 0 in some cases due to the logarithmic scale of the chart. For example, the error for two distributed children is around 50 milliseconds, which leads to a negative bar for a mean value of 10. At 100% load, the system is at its limit and the latency is significantly higher, i.e. up to two orders of magnitude. For certain runs, the mean latency is close to one 1000 milliseconds and the standard deviation indicates very inconstant results. For example, in the distributed run with three children the standard deviation shows that the many window's latency varies between 50 and 300 ms.

In Figure 31b we see that a centralized approach behaves very similarly. The latency for some centralized runs is a bit lower than that of the distributed runs, i.e. 2-5 milliseconds. This is due to the fact that once a window is complete on a centralized node, the result is immediately returned, whereas Disco first has to send it from the child node to the root node. Yet, even though latency for a centralized approach may be lower, the throughput it achieves is also significantly lower.

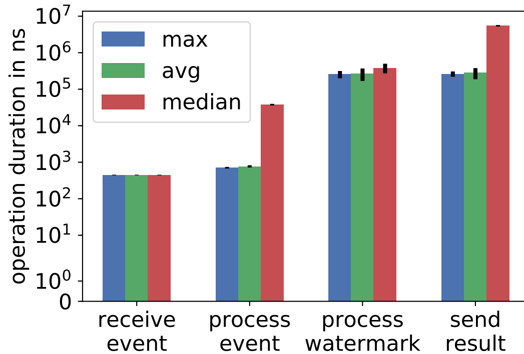


Figure 32: Individual operation duration on child node.

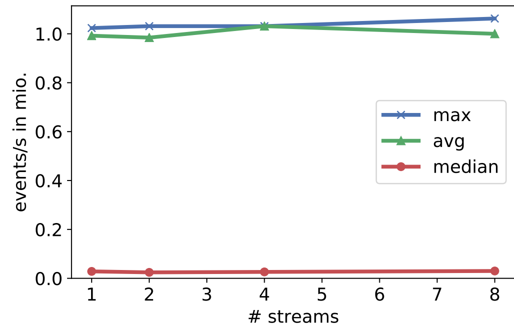


Figure 33: Increasing number of input streams for single child node.

Concluding, we see that the latency for a distributed approach is only marginally higher than for a centralized approach while having a notably higher throughput. For loads up to 75%, Disco can provide a mean window latency below 10 milliseconds.

5.2.6 Child Node Performance

In this section, we discuss the performance characteristics of a child node. We first show the duration of the individual operations on a child node in Figure 32. For these measurements, we insert multiple time measurement statements into our code around major operations. We measure the duration of receiving an event, processing an event, processing a watermark, and sending the intermediate window with its pre-aggregate. We see that receiving an event is equal for all aggregation functions, with no deviation. This is to be expected, as they all receive the same input, regardless of window and aggregate function. The first main difference we see is for processing a single event. While `max` takes 730 nanoseconds and `avg` takes 795 nanoseconds, `median`'s processing time is one and a half orders of magnitude higher at 38'000 nanoseconds. This also explains why the throughput for `median` in Figure 26c is significantly lower than for the decomposable functions. As the `combine` function for holistic functions needs to merge the two partial lists, this quickly becomes very costly for large lists because of extensive copying.

Processing a watermark and creating the final windows is again similar for all functions, as this step looks at all window metadata, determines which ones are complete, and then calculates the window aggregate. For holistic windows, we do not create the partial aggregate but assign the correct slices to the windows,

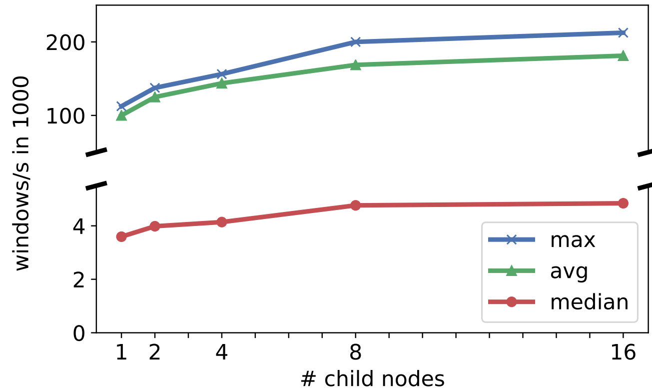


Figure 34: Processing windows at root node for increasing number of child nodes.

which performs about equally. We note here that extensive watermark checking after every single event would drastically reduce the performance of the system as, e.g. `max` currently takes 730 ns per event but the watermark check would add an additional 300'000 ns. Finally, sending the result is again significantly more expensive for holistic functions than for decomposable ones. We have to serialize the slices and actually send them, compared to simply sending some metadata and one partial aggregate value.

In Figure 33 we see that the number of input streams has no effect on the distributed aggregation. The input streams are interpreted as a single stream and are all received on the same port. This also shows that a single data generator can produce sufficient data to saturate a child node, as mentioned above. If we add up the numbers from Figure 32 for the tumbling window of one second length, used in the benchmark, we get 1.04 million events maximum possible throughput per second for `max`, 0.98 million events for `avg`, and 25'000 events for `median`, which is exactly the throughput that was achieved here.

5.2.7 Root Node Performance

In this section, we discuss the performance characteristics of the root node. We show the number of windows that the root node can handle per second with an increasing number of child nodes in Figure 34. Here, we see that the number of windows per second increases with the number of children. While for one child node we can achieve just above 100'000 windows per second, for 16 children we can achieve around 200'000 for decomposable functions. This can easily be explained by the fact that we have to `lower` fewer results for more children. When only

one child sends a window, it is automatically the final result, so each window goes through the process of computing the final aggregate and emitting the result. For n child nodes, each window needs to be received n times before the final result is ready, thus reducing the final aggregation and emitting time to $1/n$ on average per incoming window message. We see the pattern of previous results also emerges here, where `max` is slightly more efficient than `avg` because the intermediate representation is smaller and the merging process is less expensive. For the `median`, we chose slices with 1000 events each and here the receive and parse time for a single window dominate the processing time, leading to the same 1.5 - 2 orders of magnitude less throughput than for decomposable functions. This is also why the increase with a higher number of child nodes only marginally improves. In general, we see that the increase flattens at around 16 child nodes, as the cost of parsing and receiving then becomes the limiting factor.

Based on this performance, we can calculate the branching factor for the root node. For the `max` aggregation function, the root node can process just above 200'000 windows before the growth stagnates. This means, depending on the number of windows per second x that a single child node generates, we can receive data from up to $200'000/x$ children. This calculation is not expected to hold for an arbitrarily high number of child nodes, but as an intuition, if each child node can process one million events per second for `max` and the root can process 200'000 windows per second, for a simple one-second tumbling window we could scale to 200 billion events per second. This scaling factor will rapidly increase with each level of intermediate nodes that is introduced, thus making our distributed aggregation very suitable for the high volume and velocity requirements of future sensor-based IoT applications.

5.3 Discussion

In this section, we thoroughly evaluated Disco against a state-of-the-art centralized approach. Our evaluation showed that Disco can scale almost linearly with the number of nodes in the network for all three classes of aggregation functions, while a centralized approach is limited by the single node bottleneck. We achieve one million events per second per node for a single child and nearly eight million for eight children. The central approach is limited to one million per second as all events must pass through one node. We also showed that the total network cost stays nearly constant at the initial event ingest cost with increasing tree height, whereas the centralized approach's cost grows linearly with the height. This leads to an n-fold increase of the total number of raw event bytes for a height of n. For

windows that need to be processed centrally and cannot be distributed, we showed that Disco performs equally to a centralized approach. Disco can process concurrent windows at least as efficiently as a centralized approach for all three classes of aggregation functions. The latency for our distributed window aggregation is below 10 milliseconds up to a 75% load, similar to that of the centralized solution. In general, the latency of our distributed aggregation is slightly higher than that of a centralized approach as we need to transfer the windows between nodes once they are complete, compared to instantly emitting the final result on the root node. The total number of events a single node can process is not dependent on the number of incoming streams, so we can have a high branching factor at each child node up to one million produced events per second in total. The number of direct children that an intermediate node or the root node can handle easily grows up to a hundred thousand, making Disco highly scalable for future data-intensive streaming applications.

6 Related Work

Disco builds on previous research in two key areas, aggregate-sharing in stream processing and distributed aggregation in Wireless Sensor Networks. In this section, we discuss related work from both research areas. We first review aggregate-sharing in SPEs in Section 6.1 and discuss how this provides us with support for arbitrary windows. Then, we discuss aggregation approaches in WSNs and show how we can profit from distributed aggregation concepts in previously researched network structures (see Section 6.2).

6.1 Aggregate-Sharing in Stream Processing Engines

In this section, we discuss related work in the field of windowed aggregations in Stream Processing Engines. For this purpose, we first present modern SPEs followed by state-of-the-art research approaches.

Modern SPEs
We now present some modern SPEs and their approach to aggregate-sharing and windowed aggregation in general. The widely used open-source SPE Apache Flink [8] uses *Buckets*, as described in Section 2.5.3. This has the shortcoming of duplicate processing of individual events for overlapping windows. Another modern open-source SPE, Apache Storm [57], uses tuple buffers to store each incoming event separately for each window, leading to redundancy in storage and computation for overlapping windows. Also, Flink can only aggregate events for a given key on a single node, i.e. the centralized approach that we describe above. Even though this approach can be distributed by key, having skewed hot keys cannot benefit from distribution [16].

The open-source SPE Apache Spark Streaming [70] uses a slice-based approach windowing due to its micro-batching architecture. However, this limits windows in that their size and slide must be multiples of the micro-batching interval. It can thus only create static windows; session windows, for example, are not supported. Spark also provides a `treeAggregate` operation that allows for a tree-based aggregation, i.e. merging results at different levels. This is similar to our multi-level merging; however, their approach is only applicable to batch processing and not to streams and windows. We see that three of the most popular open-source SPEs all use windowing aggregation techniques that leave room for improvement regarding efficient storage and computation resource utilization.

Aggregate-Sharing in SPEs

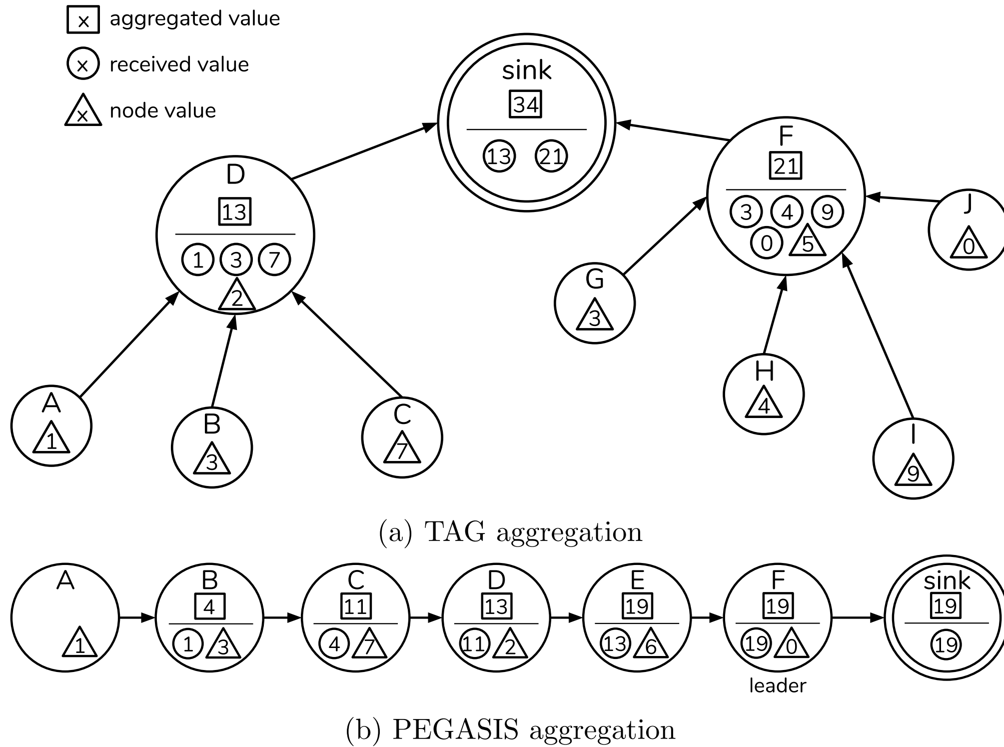
We now discuss current research focusing on this efficient computation and storage utilization. We discuss the most relevant work for our aggregate-sharing, the general stream slicing approach, in Section 2.6. As that general slicing approach by Traub et al. [58] subsumes all previous related slicing algorithms, we only briefly discuss them here.

Cutty [9] is a stream slicing algorithm that also supports arbitrary user-defined windows. When a window begins, Cutty creates a new slice, stores the running partial aggregate for the previous slice, and starts a new running aggregate. When a window is complete, the stored partial aggregates belonging to that window are merged to the final window result. However, Cutty does not support out-of-order tuples, making it unsuitable for distributed aggregation on multiple streams where out-of-order events are unavoidable due to different network characteristics.

Other stream slicing approaches are *Panes* [40] and *Pairs* [39]. Panes creates non-overlapping slices based on the window size and the window slide. It finds the greatest common divisor of the window length and slide and sets this as the slice size. Again, once a window is complete, all slices are merged with their partial aggregates. However, this approach works only on context free windows, making it unsuitable for more complex user-defined windows.

Pairs builds on the idea of Panes but does not compute the slice size by the greatest common divisor but by the modulo operation of the window range and slide. The authors show that this always leads to two slices per window, i.e. slice *pairs*. However, this approach is still only applicable to CF windows. The general stream slicing approach subsumes all previously mentioned slicing techniques and is applicable to arbitrary user-defined windows as well as out-of-order processing. While this approach supports inter-key parallelism, i.e. one slicer per key, it does not support intra-key parallelism, i.e. multiple slicers per key on different nodes, which is essential for distributed aggregation. As the focus of this work is to perform distributed aggregations, we need to extend existing solutions to support inter-key parallelism.

In conclusion, while all approaches improve the efficiency of windowed aggregation, none of them do so in a truly distributed way. Some support inter-key parallelism but none allow for intra-key parallelism, which restricts the distribution of hot keys with disproportionately many events. Disco supports intra-key distribution, leveraging the computing resources on all available nodes. We do so while still applying efficient aggregate-sharing on individual nodes and transferring unique disjoint slices between nodes to avoid redundant network traffic.



(a) TAG aggregation

(b) PEGASIS aggregation

Figure 35: Two example `sum` aggregations in WSNs.

6.2 Distributed Aggregation in WSNs

In this section, we present related work in the field of data aggregation Wireless Sensor Networks. Previous research in this area builds the foundation of the *distributed* aspect of Disco. We discuss two different network structures in detail, the tree-structured *TAG* and the chain-structured *PEGASIS*. We then also discuss the approximate aggregation approach *Sketches*. To conclude with, we discuss why we chose a tree-based structure over the alternatives and exact results over approximations for Disco.

Madden et al. [45] introduce a tree-structured in-network data aggregation method named *TAG* (Tiny AGgregation), which consists of two parts, the distribution phase and the collection phase. The distribution phase is similar to the sample routing algorithm we show in Section 2.2.1. A node starts a query by broadcasting it to its neighbors. Each neighbor stores the sender as its parent and in return broadcast the request to their neighbors, and so on, until the tree is complete. In these requests, each node also sends an interval in which it expects a value from its children. This interval decreases slightly per tree-level so that each parent has time

to collect all child values and merge them before having to provide a value for its parent. At the end of an interval, each node sends its current value to its parent, who then merges the values and passes them up the tree until it reaches the sink. This method can generally be applied to decomposable aggregation functions. The main advantage of this approach is the minimal use of bandwidth. However, should an intermediate node fail, all data of that sub-tree will be lost. The structure of *TAG* can be seen in Figure 35a. Node D and F are intermediate nodes that collect and aggregate the data of their child nodes and their own data before passing it on. Before the final value reaches the sink, the data needs to travel for only two network hops. We do not focus on the creation of the network structure in this work, but we adapt the tree-based aggregation concepts presented by Madden et al. While their network structure is similar to our approach, they support only tumbling windows and decomposable aggregation functions. In this thesis, we extend the tree-based approach to support arbitrary windows and aggregation functions.

Cougar [22, 65] and *LEACH* [25] are both alternative tree-based approaches that focus on data aggregation at *cluster* nodes. The main concepts behind their work deal with the election of such cluster nodes in WSNs. Another related tree-based approach is Directed Diffusion [32], where the authors build routing trees for multiple sinks by broadcasting the queries and back-tracking the most efficient path from sensor to sink. Our work, on the other hand, runs in a Fog network structure, where the cluster nodes are naturally given through the network's hierarchy. Thus, we do not need to deal with efficient paths in networks of equal nodes. Similarly to TAG, Directed Diffusion, Cougar, and LEACH can all only process simple tumbling windows. Also, only Cougar deals with more complex aggregation functions. We extend the authors' ideas to support arbitrary window types and complex data aggregation.

Another distributed data aggregation approach is *PEGASIS* [43], which creates a chain of nodes in the network. For this, each node needs to know the location of all other nodes. Each node can then compute the same global chain of nodes, based on a greedy algorithm. The algorithm selects the closest unvisited neighbor for each node as the node's successor. As the algorithm always begins with the node farthest from the sink, the result is deterministic on all nodes. The data is then sent along this chain and merged (in our case aggregated) at each node. The last node in the chain (leader) then sends the final result to the base station or out of the system. The leader changes in each iteration to evenly distribute the energy load on all devices. Should a node fail, the algorithm simply determines the next hop in the chain as the new successor and thus avoids a partitioned network. This method can generally be applied to decomposable aggregation functions. PEGASIS provides fault-tolerance through the global knowledge of the network

at each node. However, this comes at the cost of reduced latency, as a message has to take n hops (n being the number of nodes) until it reaches the sink. The *PEGASIS* approach can be seen in Figure 35b, where the chain forms from node A to F. A sends its data to B, who adds A's value to its own $(1 + 3)$ and passes on 4 to C. In this example, until the final vale reaches the sink, it has to take six network hops, compared to two in *TAG*. We chose a tree-based approach similar to TAG or Cougar due to the natural tree-like structure of Fog networks, in order to avoid global state at multiple nodes and to reduce the latency of our aggregation results.

Sketches [11] represents a class of approximate distributed aggregation algorithms. In this approach, Considine et al. adapt counting sketches [15] to support `sum` and thus also `average` aggregations (average can be computed with count and sum). Sketches are bitmaps of a fixed length with a binary hash function $h()$. The result of the hash function is then used to set the bit at a certain position in the bitmap to 1 for a given value. We note here that the authors propose multiple functions by which a bit is flipped and thus, we only provide an intuition behind the concept. The goal of such a function is to evenly distribute the positions of the bits to flip. This can then be used to approximate the number of values inserted into the sketch by looking at the probability of each bit flipped. The authors use a function where a bit z can only be set to 1 if the value to be inserted is greater than or equal to z. Now looking at the combination of bits set to 1 and their positions, the sum can be approximated.

For *Sketches*, the actual aggregation algorithm in the WSN uses this sketch approach. The query is broadcast by the sink and then broadcast by all its neighbors, and so on until all nodes have received the query. Each node stores its level, that of its direct neighbors and a sketch containing its one value. Now starting from the lowest level (farthest from the sink), the nodes broadcast their sketches to their neighbors that are one level higher. The higher nodes then insert the received values into their sum-sketch and in return broadcast their sketches one level up again. As the sketches are duplicate insensitive, processing the same value multiple times is not a problem and increases fault-tolerance. Finally, the sink receives its neighbors' sketches and can compute the final value.

This approach is applicable only to basic aggregation functions, i.e. `sum`, `count`, and `average`. As the query is broadcast in levels, the number of network hops, until all values have reached the sink, is expected to be in $O(log\ n)$, where n is the number of nodes. Even though Sketches provides fault-tolerant aggregation, the results are only approximate and not suitable for many streaming applications due to strict business logic. Also, this approach supports only the basic aggregations

`count`, `sum`, and `avg`, which is not sufficient for more complex analysis. Thus, we chose an exact-result aggregation as proposed in the tree-based approaches above.

In summary, many data aggregation or reduction techniques have been researched in the field of Wireless Sensor Networks. However, none of them focus on complex windowing semantics and only a few take different aggregation functions into account. In our work, we aim to combine data reduction techniques from WSNs, i.e. partial aggregation, with complex business logic required by modern streaming processing applications. Thus, Disco differs from all previously mentioned WSN approaches in that we support arbitrary windows and aggregation functions.

7 Conclusion

In this thesis, we presented Disco, a stream processing framework for distributed data aggregation on arbitrary windows. Disco combines the distributed data aggregation concepts from Wireless Sensor Network with the complex windowing and aggregation semantics from modern Stream Processing Engines. This allows us to benefit from reduced network traffic through node distribution in Fog networks while providing efficient aggregations on arbitrary windows.

We first discussed the foundations of Disco, namely Fog computing, WSNs, and SPEs. We then presented a general classification of streaming windows and aggregation functions, followed by the general stream slicing approach, which is a core component of Disco. For our distributed window aggregation we first introduced the theoretical concepts of distributed windows and their interpretations. We discussed how context free, forward-context free, and forward-context aware windows can generally be viewed in a distributed setting while showing how they can be merged on independent nodes and transferred efficiently between them. This was followed by a detailed account of our implementation of these concepts. We described the three core components of Disco, the child node, the intermediate node, and the root node and how they interact with each other.

In our evaluation, we showed the advantages of a distributed window aggregation compared to a central one. Disco scales linearly with the number of nodes that process incoming events while supporting inter- and intra-key parallelism. The network cost of our distributed aggregation is nearly constant compared to a linear cost increase for a centralized approach. We show that the performance of Disco for multiple different windows and aggregation functions is at least as good as in the central approach, even for settings that require central computation. Also, our evaluation shows that our aggregation tree can grow up to hundreds of thousands of nodes, making Disco incredibly scalable and suitable for future data-intensive distributed streaming applications.

Our focus in this thesis was on the support of complex aggregation functions and window types. A lot of research questions are posed by our distributed aggregation approach, which we could not address in this work. A central open question is that of fault-tolerance. We aggregate data in a tree-based structure, meaning individual intermediate nodes become single points of failure for their entire sub-tree. Also, further research can look into reducing the size of individual slices that need to be sent between nodes, e.g. by applying common compression techniques. Currently, Disco requires a static network topology. Interesting open questions around a

dynamic network structure also need to be investigated in the future, i.e. what happens if nodes enter and leave the network.

In this work, we have shown that complex data stream analysis can benefit from distributed aggregation. We believe that modern SPEs will soon adapt to support distributed deployments outside of data centers and that new Fog-aware SPEs will emerge to tackle the ever-increasing data volume and velocity.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[2] Ian Fuat Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[3] Arvind Arasu and Jennifer Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, pages 336–347. VLDB Endowment, 2004.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16. ACM Press, 2002.

[5] Stephanie B. Baker, Wei Xiang, and Ian Atkinson. Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities. *IEEE Access*, 5:26521–26544, 2017.

[6] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to Rule Them All. In *SIGMOD*, pages 1757–1772. ACM, 5 2019.

[7] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart Cities in Europe. *Journal of Urban Technology*, 18(2):65–82, 2011.

[8] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache Flink(TM): Stream and Batch Processing in a Single Engine. *Bulletin of the Technical Committee on Data Engineering*, 38(4):28–38, 2015.

[9] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*, pages 1201–1210. ACM, 2016.

[10] Louis Columbus. 2018 Roundup Of Internet Of Things Forecasts And Market Estimates. `https://www.forbes.com/sites/louiscolumbus/2018/12/13/2018-roundup-of-internet-of-things-forecasts-and-market-estimates/`. Accessed: 2019-09-26.

[11] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, pages 449–460. IEEE Comput. Soc, 2004.

[12] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8):112–116, 2016.

[13] Yao-Chung Fan and Arbee L.P. Chen. Efficient and robust sensor data aggregation using linear counting sketches. In *IPDPS*, pages 1–12. IEEE, 2008.

[14] Elena Fasolo, Michele Rossi, Jorg Widmer, and Michele Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *IEEE Wireless Communications*, 14(2):70–87, 2007.

[15] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[16] Improvement to Flink Window Operator with Distributed Aggregation. https://cwiki.apache.org/confluence/display/FLINK/FLIP-44%3A+Support+Local+Aggregation+in+Flink. Accessed: 2019-10-22.

[17] Improvement to Flink Window Operator with Slicing. http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-Improvement-to-Flink-Window-Operator-with-Slicing-td25750.html. Accessed: 2019-09-11.

[18] Improve performance of Sliding Time Window with pane optimization. https://issues.apache.org/jira/browse/FLINK-7001. Accessed: 2019-09-11.

[19] Poor performance with Sliding Time Windows. https://issues.apache.org/jira/browse/FLINK-6990. Accessed: 2019-09-11.

[20] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. Coordinated Omission in NoSQL Database Benchmarking. In *BTW*, pages 215–225. Gesellschaft für Informatik, 2017.

[21] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[22] Johannes Gehrke and Samuel Madden. Sensor and actuator networks - Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004.

[23] Tuan Nguyen Gia, Mingzhe Jiang, Amir-Mohammad Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. Fog Computing in Healthcare Internet of Things: A Case Study on ECG Feature Extraction. In *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 356–363. IEEE, 2015.

[24] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[25] Wendi Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, 2002.

[26] Wendi Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *MobiCom*, pages 174–

185. ACM, 1999.

[27] Thomas Hießl, Christoph Hochreiner, and Stefan Schulte. Towards a Framework for Data Stream Processing in the Fog. *Informatik Spektrum*, 42(4):256–265, 2019.

[28] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):1–34, 2014.

[29] Huiping Huang, Shide Xiao, Xiangyin Meng, and Ying Xiong. A Remote Home Security System Based on Wireless Sensor Network and GSM Technology. In *NSWCTC*, pages 535–538. IEEE, 2010.

[30] Mark Hung. *Leading the IoT - Gartner Insights on How to Lead in a Connected World*. Gartner Research, 2017.

[31] Kai Hwang, Yue Shi, and Xiaoying Bai. Scale-Out vs. Scale-Up Techniques for Cloud Performance and Productivity. In *CloudCom*, pages 763–768. IEEE, 2014.

[32] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, 2003.

[33] Namit Jain, Stan Zdonik, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, and Richard Tibbetts. Towards a streaming SQL standard. *PVLDB*, 1(2):1379–1390, 2008.

[34] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 grand challenge. In *DEBS*, pages 266–269. ACM Press, 2014.

[35] Paulo Jesus, Carlos Baquero, and Paulo Sergio Almeida. A Survey of Distributed Data Aggregation Algorithms. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2015.

[36] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking Distributed Stream Processing Engines. In *ICDE*, pages 1507–1518. IEEE, 2018.

[37] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*, pages 555–569. ACM, 2016.

[38] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *NetDB*, pages 1–7, Athens, Greece, 2011. ACM.

[39] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*, pages 623–634. ACM, 2006.

[40] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[41] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Seman-

tics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322. ACM Press, 2005.

[42] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing. *PVLDB*, 1(1):274–288, 2008.

[43] Stephanie Lindsey, Cauligi Raghavendra, and Krishna Sivalingam. Data gathering algorithms in sensor networks using energy metrics. *TPDS*, 13(9):924–935, 2002.

[44] Guojin Liu, Rui Tan, Ruogu Zhou, Guoliang Xing, Wen-Zhan Song, and Jonathan M Lees. Volcanic Earthquake Timing Using Wireless Sensor Networks. In *IPSN*, pages 91–102. IEEE, 2013.

[45] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, pages 131–146. ACM, 2002.

[46] Amit Manjhi, Suman Nath, and Phillip B Gibbons. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. In *SIGMOD*, pages 287–298. ACM, 2005.

[47] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn Mckinley, and Felix Xiaozhu Lin. StreamBox: Modern Stream Processing on a Multicore Machine. In *ATC*, pages 617–629. USENIX Association, 2017.

[48] Carla Mouradian, Diala Naboulsi, Sami Yangui, Roch Glitho, Monique Morrow, and Paul Polakos. A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges. *IEEE Communications Surveys & Tutorials*, 20(1):416–464, 2018.

[49] Snehal Nagmote and Pallavi Phadnis. Massive Scale Data Processing at Netflix using Flink. Flink Forward San Francisco, 2019.

[50] Suman Nath, Phillip Gibbons, Srinivasan Seshan, and Zachary Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. In *SenSys*, pages 250–262. ACM, 2004.

[51] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *PVLDB*, 10(12):1634–1645, 2017.

[52] Emil Petriu, Voicu Groza, Dimitrios Makrakis, Dorina Petriu, and Nicolas Georganas. Sensor-based information appliances. *IEEE Instrumentation & Measurement Magazine*, 3(4):31–35, 2000.

[53] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.

[54] Apache Spark the fastest open source engine for sorting a petabyte. `https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`. Accessed: 2019-09-06.

[55] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[56] Gil Tene. How NOT to measure latency. `https://www.infoq.com/presentations/latency-response-time/`, 2016. Accessed: 2019-10-17.

[57] Ankit Toshniwal, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, and Maosong Fu. Storm@twitter. In *SIGMOD*, pages 147–156. ACM Press, 2014.

[58] Jonas Traub, Philipp Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Efficient Window Aggregation with General Stream Slicing. In *EDBT*, pages 97–108. OpenProceedings, 2019.

[59] Jonas Traub, Philipp Grulich, Alejandro Rodriguez Cuellar, Sebastian Bress, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Scotty: Efficient window aggregation for out-of-order stream processing. In *ICDE*, pages 1304–1307. IEEE, 2018.

[60] Peter Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

[61] Luis Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50, 2008.

[62] Ashish Vulimiri, Carlo Curino, Thomas Jungblut, Jitu Padhye, George Varghese, and Implementation Nsdi. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *NSDI*, pages 323–336. Usenix Association, 2015.

[63] Bjourn Wiedersheim, Michel Sall, and Guillaume Reinhard. SeVeCom — Security and privacy in Car2Car ad hoc networks. In *ITST*, pages 658–661. IEEE, 2009.

[64] Alec Woo and David Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *MobiCom*, pages 221–235. ACM, 2001.

[65] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

[66] Shanhe Yi, Cheng Li, and Qun Li. A Survey of Fog Computing. In *Mobidata*, pages 37–42. ACM Press, 2015.

[67] Young Han Nam, Zeehun Halm, Young Joon Chee, and Kwang Suk Park. Development of remote diagnosis system integrating digital telemetry for medicine. In *IEMBS*, volume 3, pages 1170–1173. IEEE, 1998.

[68] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey. *CoRR*, abs/1808.05283:0–48, 2018.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, and Scott Shenker. Spark: Cluster Computing with Working Sets. In *HotCloud*, pages 1–7. USENIX Association, 2010.

[70] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and

Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM Press, 2013.

[71] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing Efficient Stream Processing on Modern Hardware. *PVLDB*, 12(5):516–530, 2019.

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used. The independent and unaided completion of this thesis is affirmed by affidavit:

Potsdam, 01.11.2019

Lawrence Benson