Deco: Fast and Accurate Decentralized Aggregation of Count-based Windows in Large-scale IoT Applications

Wang Yue
HPI & U Potsdam

Rafael Moczalla
HPI & U Potsdam & IAV

Manisha Luthra
TU Darmstadt & DFKI

Tilmann Rabl
HPI & U Potsdam

ABSTRACT

In the realm of large-scale Internet-of-Things applications, aggregating data using count-based windows is a formidable challenge. Current methods, either centralized and slow or decentralized with potential inaccuracies, fail to strike a balance. This paper introduces Deco, a novel approach tailored for swift and precise aggregation in distributed stream processing systems. Accomplishing this balance is complex due to the dynamic nature of event distribution: events arrive at varying rates, unordered, and at diverse times, making *accurate* window computation a challenge. To overcome this, we propose a lightweight *prediction method* that derives local window sizes based on the previously observed event rates and performs corrections when necessary to ensure *accurate* and *fast* query results. These windows are processed in a decentralized manner on local nodes, verified for correctness and then aggregated on a root node. Our evaluation showcases Deco's superiority over centralized methods, outperforming others significantly. Deco reduces network traffic by up to 99% and exhibits linear scalability with node count.

1 INTRODUCTION

IoT forms huge decentralized networks. Internet of Things (IoT) applications are omnipresent in many domains [20, 55] that range from environment monitoring [39] and industry 4.0 [61] to health care [13]. By 2025, the expected number of IoT devices enabling such applications is expected to surpass 75 billion [20]. This massive amount of devices forms huge decentralized networks and generates continuous high-speed data streams. Current Stream Processing Engines (*SPEs*) such as Flink [9], Spark Streaming [66], and Kafka Streams [43, 56] serves as a natural fit to process such large-scale and high-velocity data streams arriving from IoT devices because of their inherent capabilities of dealing with distributed unbounded data streams [32].

Typically, SPEs bound the incoming data streams from IoT devices to fixed-sized data sets called windows (either time- or count-based). These windows are often aggregated with a function such as an *average* or a *sum*. Existing SPEs aggregate such events from different IoT data sources on a centralized node, which can ultimately become a bottleneck for large-scale decentralized IoT networks. Recent work such as NebulaStream [67], Disco [6], and Desis [64] propose a distributed window aggregation approach, where multiple nodes process windows and aggregations. These approaches are restricted to time-based windows where the data stream can be easily split based on time, e.g., multiple nodes can process equally sized time windows.

Count-based windows, instead, require data elements to be accumulated based on a fixed number of events, e.g., 1 million events per window, which is why splitting count-based windows for decentralized aggregation becomes *problematic*. This is because splitting count-based windows requires prior information on the incoming event rates from the local nodes at a central node to ensure correctness and fairness in the distribution but processing centrally can easily become a bottleneck in large-scale decentralized IoT networks. This is not a one-time effort because, in stream processing, applications often observe non-uniform distributions where event rates change continuously, and thus fetching event rates becomes particularly hard.

The process is further complicated when partial aggregates are to be combined and the correctness of combined aggregates needs to be ensured while dealing with unordered and late events. Thus, segregating and calculating count-based window aggregates on multiple nodes is a challenge for large-scale decentralized IoT networks as we motivate next with an IoT example.

Motivating example. In the context of a smart factory setup, we encounter significant challenges related to real-time data processing. In this scenario, local nodes are distributed across different locations in the factory and continuously collect data about various assembly line activities from weak sensors. These data streams include information about the *number* and *types of products* being manufactured, crucial for ensuring the *right* quantity and quality of the products. One of the key tasks involves aggregating counts of specific events, such as finding the minimum, maximum, or average quality of products within batches. This process makes a count-based window aggregation *essential* for monitoring the quality and efficiency of the assembly lines. However, this seemingly straightforward task becomes complex due to the dynamic nature of the data streams. The challenge arises from the fact that the event rates, indicating how fast these products are being produced, are not constant; they change mildly but frequently based on the assembly line's speed and product demand. There might be delays in reporting products depending on the assembly schedule, leading to unordered or late events.

To address this issue, one solution is to use approximate event rates, which can help in splitting time windows and processing events in a timely manner. However, relying solely on approximations [3, 16, 24, 34, 35, 54] can lead to errors. In a setting where product batches are subject to rigorous quality control, such errors are unacceptable. In this scenario, the challenge is to find a balance between processing data swiftly and accurately.

Traditional centralized techniques [6, 9, 43, 56, 64, 66] are accurate but can become slow and inefficient as the scale of the monitoring system grows to handle millions of assemblies. Existing decentralized techniques, while fast, often deliver incorrect results due to the approximations made. This dilemma highlights the crucial need for advanced solutions that can navigate the complexities of varying event rates and delays while ensuring both speed and accuracy in data processing.

Deco: a decentralized aggregator for count windows. In this work, we present Deco, a DEcentralized approach for calculating COunt-based window aggregations on multiple nodes by utilizing event rate information rather than aggregating all events on the centralized node and ensure fast and correct results. The key idea of Deco is to *predict* window sizes based on the incoming event distribution previously observed. These predictions are used by the local nodes for window processing, where events are partially aggregated. The partial aggregates are delivered to a root node where the final aggregation is computed (Figure 1). Verification and correction steps at the local nodes verify if the predictions match the actual values otherwise update (correct) the predictions to ensure the correctness of results.

We propose three schemes of Deco that differ in utilizing the observed event rates for splitting and computing aggregation (Deco_{mon}) instead of predictions for local windows, and using synchronous ($\text{Deco}_{\text{sync}}$) or asynchronous ($\text{Deco}_{\text{async}}$) communication schemes to distribute calculations. We discuss the differences between these schemes and show that Deco outperforms state-of-the-art centralized techniques like Scotty [60] and distributed approaches like Disco [6] for count-based windows by an order of magnitude. Deco is most efficient when event rates per data source are stable, while it also can adapt to changing event rates. Deco processes count-based windows decentralized with the same results as centralized solutions.

Summary of contributions. (1) We propose Deco, which enables decentralized window aggregation on distributed streams. (2) We present three schemes, Deco_{mon}, $\text{Deco}_{\text{sync}}$, and $\text{Deco}_{\text{async}}$. Deco_{mon} moves partial aggregations to nodes closer to the data sources based on observed rates and local calculation of window sizes. $\text{Deco}_{\text{sync}}$ and $\text{Deco}_{\text{async}}$ reduce the communication overhead in local window calculation using predictions. $\text{Deco}_{\text{sync}}$ blocks nodes during communication while $\text{Deco}_{\text{async}}$ uses asynchronous. (3) We extensively evaluate Deco and show that it outperforms state-of-the-art centralized approaches by orders of magnitude with respect to throughput and network utilization. The code on our implementation is available on GitHub[1].

The rest of the paper is structured as follows. Section 2 gives the necessary background. Section 3 presents the Deco system model. Section 4 discusses the technical details of Deco. We evaluate Deco in Section 5 and discuss related work in Section 6.

2 BACKGROUND

To process streamed queries on unbounded data streams, SPEs group events into bounded windows and execute aggregation functions on windows to output results. In this section, we introduce the foundation of window aggregation and then discuss why window aggregation in centralized setting is a bottleneck.

2.1 Window Types

Based on the window type an SPEs divides a potentially infinite data stream into finite windows. There are three major window types that are supported by existing SPEs: tumbling windows, sliding windows, and session windows [1]. A tumbling window divides data streams into event groups of L successive events. The sliding window is defined with a fixed length L and a step S. S determines the count offset between the start of the current window and the start of the previous window and L is the window size. In contrast to that, a session window is terminated by a gap in which no events arrive for a fixed amount of time. Examples of session windows are HTTP sessions and ATM interactions. In addition, there are user-defined windows [10] that are not in Dataflow and they start and end with user-defined events. Tumbling and sliding windows are created and terminated periodically and have fixed window sizes. Session and user-defined windows have unfixed window sizes since their window lengths are determined by session gaps and user-defined events.

2.2 Window Measures

Window measures decide how to measure the size of a window. There are two window measures: time and count. For time-based windows, the window start and end are determined by time, e.g., output average temperature every 5 seconds. Even though every window has the same window size, they can hold different number of events. For count-based windows, the window start and end are determined by the number of events in a window, e.g., output average temperature of the last 1000 manufactured products. The time span of count-based windows varies and relies on the generation speed of events from data streams but each window has the same number of events. In this paper, we focus on count-based windows.

2.3 Aggregation Function

In Data Cube [28], Gray et al. categorize aggregation functions into three types: distributive, algebraic, and holistic. Distributive aggregate functions can perform partial aggregation on a subpart of a dataset and then merge partial results of all sub-parts to output the results of the function, e.g., *sum*, *count*, and *min*. Algebraic aggregate functions can be computed from results of distributive aggregate functions, e.g., *avg* (as *sum* / *count*). Holistic aggregate functions cannot be calculated by partial aggregation, e.g., *median* or *quantiles*. Jesus et al. [36] similarly categorize aggregation function as decomposable, self-decomposable, and non-decomposable. Self-decomposable and decomposable functions can split windows into slices with finer granularity. Events in slices are partially aggregated and final results are computed by combining partial results of the slices. Non-decomposable functions cannot split windows in advance.[2] In this work, we focus on windows with (self-)decomposable functions.

2.4 Centralized Aggregation is a Bottleneck

Existing methods often resort to centralized aggregation for processing count-based windows, consolidating data from diverse sources into a central node. [9, 43, 56, 66]. While this approach guarantees the accuracy of query results, it presents a significant challenge. As the number of data sources and events per source increases, a common scenario in large-scale IoT applications (as highlighted in the earlier example), the central aggregator can become a bottleneck. This occurs because all the data is transmitted over the network and processed centrally, causing the central node to slow down and potentially increase network costs as the network expands. In Deco, we aim to address this challenge by decentralized aggregation while offering the same accuracy in the query results.

3 DECO SYSTEM MODEL

Data Stream Model. In decentralized deployments, data streams are produced from distributed nodes as seen in Figure 1. Those datastream nodes are weak sensor nodes and only produce data. A stream is an infinite series of tuples $t \in s$. A tuple, i.e., data event, $t = (i, v, \tau)$, value v, timestamp $\tau \in \mathbb{N}^+$ and id i are assigned by the data stream node. [12] All events are produced in order per sensor, so their timestamps monotonically increase. Deco uses timestamps and watermarks for event ordering like

[1]https://github.com/hpides/Deco#deco

[2]Deco performs centralized aggregation for non-decomposable functions.

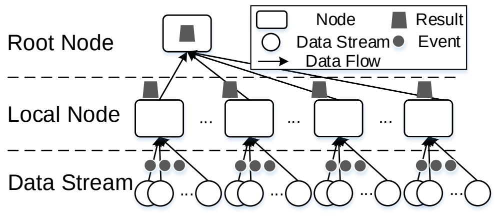


Figure 1: Network of a decentralized streaming topology

other SPEs including Flink. The datastream nodes produce events at an *event rate* that is defined as the number of events received per second. We distinguish the event rates of local nodes and data stream nodes (directly from the sensors). The global event rate is the sum of the event rates of all local nodes.

Window Operator Model. Once a window starts, we put events into the window. When the window ends, we aggregate the value of each event to produce the result. Commonly, the result is a number. Additionally, we use a stable sorting algorithm, and events are ordered by timestamp in windows. When two events share the same timestamp at the count-based window edge, we use the first one. Figure 1 shows the topology of decentralized networks. Data streams are distributed onto local nodes (middle layer) followed by final aggregation on the root node (top layer). We denote the top node as a *root node* and middle devices as *local nodes*. Local nodes are assumed to be wimpy but smart devices, e.g., edge switches [18] and routers [19] that can store a window of up to 1 million events (33 MB of RAM in our experiments (Section 5)). The root node is assumed to be more powerful (cloud server) and can process partial results from local nodes.

We distinguish windows calculated on root and local nodes as *global window* and *local window*, respectively. A local window serves as a subset of events of the global window. We refer to every count-based window as a *global window*, e.g., the global window size is 1 million when the query is to output a result of every 1 million events. The global window size is given by the query and is not changeable. In centralized aggregation, the local nodes only forward the raw events to the root. Therefore, no local windows are formed and we have only a global window that is processed at the root node. In decentralized window aggregation, instead, a global window comprises a set of local windows, where events are partially aggregated on local nodes and forwarded to the root node for the final aggregation.

Communication Model. When processing windows in decentralized deployments, local nodes have to receive messages from the root node, e.g., window types, window measures, and aggregation functions. Also, the root node receives messages from local nodes, e.g., partial results, raw events, and event rates. We define the communication between the local and the root node as a *flow*. It refers to a single-direction communication, e.g., information that is sent from the local node to the root or the answer is sent back from the root back to the local node. We denote *up-flow* as the communication from local nodes to root and *down-flow* as the one from root to local nodes. Current centralized approaches are single-flow approaches since there is only one up-flow communication for each global window.

4 THE DECO SYSTEM

In this section, we first introduce how the decentralized aggregation model works and identify the issues Deco approach has to deal with. We then present the architecture and technical details of the Deco and discuss optimization ideas for various situations.

4.1 Approximate Decentralized Aggregation

To process count-based windows locally, we let the root node coordinate local nodes, i.e., we execute count-based window operators on the local nodes. The root node determines for each local node the local window size l such that windows can terminate locally. The local nodes need global information to determine the correct end of a local count-based window. In real-world applications, the event rate f of each source can be different. We calculate local window sizes by analyzing the local event rates of data streams. In a decentralized network, there are many data streams and the number of streams connected to each local node is also different. We formally define the theoretical local window size as follows. Given a local node a, let n be the number of connected data streams. Let $f_1, f_2, \ldots, f_n$ be the event rates of all connected data streams where f refers to the event rate of a single stream. Then $f_a = \sum_{i=1}^{n} f_i$ is the total event rate of node a. For a decentralized network with m local nodes and the total event rate of the root node is $f_{root} = \sum_{i=1}^{m} f_i$. Furthermore, let l_{global} be the global window size and $l_a = \frac{f_a}{f_{root}} * l_{global}$ be the local window size of node a. The global window size is equal to the sum of all local window sizes $\sum_{i=1}^{m} l_i$ comprising events of this global window. For example, for two local nodes a and b with a global window size of 1 million events, where the local event rate of node a is 1.2 million events per second and of node b is 0.8 million events per second. The local window size of node a will be determined as $\frac{1.2}{(1.2+0.8)} * 1$ million, which is equal to 0.6 million events and similarly for local node b is 0.4 million events.

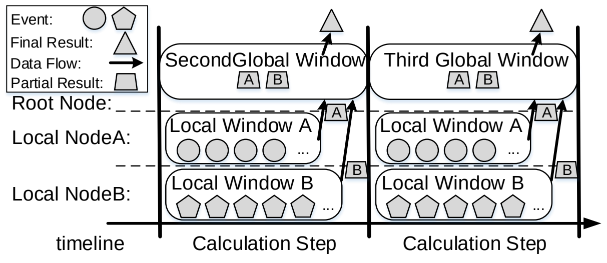


Figure 2: Approximate decentralized aggregation

A naive approximate decentralized aggregation approach has two steps, *(i)* initialization step, and *(ii)* calculation step. When processing the first global window, the root node executes the initialization step that creates a global window and collects all events from local nodes. Once the global window ends, the root aggregates events and outputs the result. Also, local window sizes (l_i) are calculated and sent to the local nodes. From the second global window onward, only the calculation step is executed, as shown in Figure 2. Local nodes use local window sizes to create local windows and send partial results to the root node. In the root node, only partial results are collected and processed. There is only one up-flow in each global window and every local window reuses the same local window size as the previous one. The single-flow approach applies decentralized aggregation to process count-based windows, so windows can be aggregated locally and they send only the partial results instead of raw events.

This simple approach in general does not produce correct results in the case of changing event rates. A local node might contribute more or fewer events to the global window. This leads to varying sizes of local windows because the global window size is fixed, so an increase in one local window size causes another local window size to decrease. Therefore, when the event rate changes and the partial result is still calculated with the static local window size, the final result is incorrect. In the following, we discuss how Deco approaches address these issues.

4.2 Deco Approaches

In the following we explain our three approaches that are capable of adapting to changing event rates. The Deco approach ensures that the results are similarly correctly delivered like centralized aggregation solutions. In decentralized aggregation, Deco splits windows and moves window aggregations to local nodes and dynamically adapts to changing event rates. This is done by recomputing local window sizes for each global window as the event rates of each data stream change.

4.2.1 Monitoring scheme: Deco$_{mon}$. In order to adapt to changing event rates, local nodes recalculate local window sizes for each new global window. In Figure 3, we show the mechanism of Deco$_{\text{mon}}$, which uses monitoring to compute local window sizes and distribute window computations to local nodes. For each global window, Deco$_{\text{mon}}$ performs a sequence of three steps: (1) initialization, (2) verification, and (3) calculation. During the initialization step, all local nodes send measured event rates from all the data sources to the root node. Once the event rates are collected from all data sources, the system starts the verification step. The root node calculates local window sizes ($l_{a,G1}$ for node a and $l_{b,G1}$ for node b) of the global window ($G1$) and sends them back to local nodes. During the calculation step, local nodes create local windows based on the local window sizes received from the root and perform the local window aggregation.

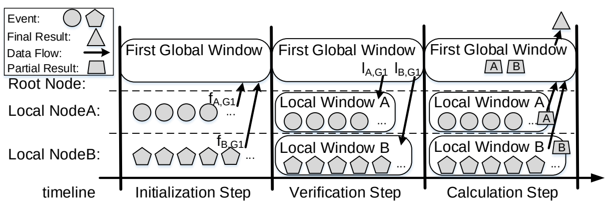


Figure 3: Deco$_{\text{mon}}$ scheme

Compared to centralized aggregation, the Deco$_{\text{mon}}$ can offload computations to local nodes and reduce the network overhead of a decentralized setup. This is because local nodes only send partial results instead of events to the root node. However, it is a synchronized approach since the root node starts the verification step only if it receives the event rate from local nodes and local nodes start calculating results when they have local window sizes. In this case, for every global window, there are three communication flows between the root and local nodes. That leads to significant latency.

To reduce communication flows, we can merge initialization step and calculation step of two consecutive global windows as they are both consecutive up-flows. The root node does not wait for the initialization step to finish but computes local window sizes based on the event rates of the last global window and assigns them to local nodes. In this case, the root node will produce wrong local window sizes because the event rates sent by local nodes are outdated when the event rates change, so the results will be wrong. Therefore, in Deco$_{\text{mon}}$ we cannot merge different steps without losing correctness. In the following section, we propose another scheme that reduces communication flows and still emits correct results.

4.2.2 Synchronous Scheme: Deco$_{sync}$. To reduce communication flow and have correct results, we propose Deco$_{\text{sync}}$ which applies a simple prediction technique for local window sizes. In general case, Deco$_{\text{sync}}$ has three steps, i.e., prediction, calculation, and verification. Deco$_{\text{sync}}$ is a synchronized approach. The root node predicts the local window sizes. If the prediction is wrong, Deco$_{\text{sync}}$ will perform a correction step to produce the correct results.
Initialization. For the first two global windows, Deco$_{\text{sync}}$ has no event rates that can be used for prediction. Thus for them, the root node collects all events from the local nodes in a centralized fashion and computes the global window. The root node applies the aggregation functions and emits the result once the window ends. When the second window ends, the root node calculates the local window sizes for each local node. We denote the local window sizes of node a for the first and second global windows as $l_{a,G1}$ and $l_{a,G2}$.
Prediction. From the third window onward, Deco$_{\text{sync}}$ predicts the local window sizes to push down the count-based window operator to local nodes and aggregate partial windows there. In this step, local nodes wait until the message from the root node. To distinguish between the actual local window size and predicted local window size, we let the predicted local window size of the node a in the third global window be $\hat{l}_{a,G3}$. We reuse the actual local window size of the previous window as the predicted local window size of the window. We present Algorithm 1 that shows the prediction step.

Algorithm 1 Deco$_{\text{sync}}$ performs the prediction step of window i on the root node.

```
Input: $l_{a,Gi-1}$, $l_{a,Gi-2}$: last two windows
Output: $\hat{l}_{a,Gi}$, $\Delta_{a,Gi}$: predicted local window and delta as small buffer
1: $\hat{l}_{a,Gi} = l_{a,Gi-1}$
2: $\Delta_{a,Gi} = |l_{a,Gi-1} - l_{a,Gi-2}|$
3: send $\hat{l}_{a,Gi}$ and $\Delta_{a,Gi}$ to local nodes
```

We assume that event rates change slightly and the local window sizes in two consecutive global windows are close. The predicted local window size is defined as follows.

$$\hat{l}_{a,Gi} = l_{a,Gi-1}. \tag{1}$$

However, the predicted local window size is not the actual local window size when the event rates vary. We introduce a factor delta (Δ) to enable the root node to correct prediction errors of local window sizes. The *delta* of a local node a in third global window is computed as $\Delta_{a,G3} = |l_{a,G2} - l_{a,G1}|$. It reflects the difference between the consecutive windows, so we only look at the delta between the past two local windows. The delta for a given node a for any global window after the second global window is defined as follows:

$$\Delta_{a,Gi} = |l_{a,Gi-1} - l_{a,Gi-2}|. \tag{2}$$

Example: For ease of understanding, here is a numerical example. The local window size of node a in the last two global windows are 0.6 million ($l_{a,Gi-2}$) and 0.601 million ($l_{a,Gi-1}$). Then, $\hat{l}_{a,Gi}$ will be 0.601 million and $\Delta_{a,Gi}$ will be 1000. Then, the root node assigns all predicted local window sizes and deltas to their corresponding local nodes. Different nodes have different predicted window sizes and deltas. The prediction step ends and the root node waits for messages from local nodes.

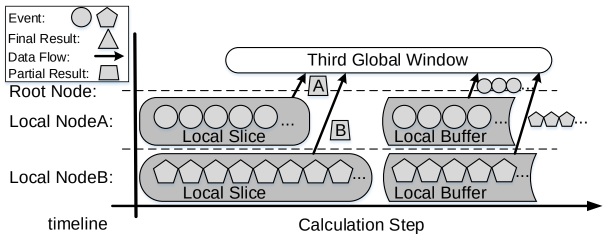


Figure 4: Calculation step in local nodes of Deco$_{sync}$.

Calculation. In the calculation step, the root node waits, and local nodes create windows and process events. Local nodes create local windows after receiving predicted local window sizes and deltas from the root node as shown in Figure 4. Deco$_{sync}$ creates a local window on node a and divides the window into two parts, a local slice and a local buffer. We present Algorithm 2 on the calculation step.

Algorithm 2 Deco$_{sync}$ performs calculation step of window i on the local node a.

```
Input: $\hat{l}_{a,Gi}$, $\Delta_{a,Gi}$: predicted window size and delta
Output: $partial\_result, buffer[l_{a,buffer}], event\_rate$
1: $l_{a,slice} = \hat{l}_{a,Gi} - \Delta_{a,Gi}$
2: $l_{a,buffer} = 2 * \Delta_{a,Gi}$
3: $slice[l_{a,slice}] \leftarrow events$                          ▷ read events
4: $partial\_result \leftarrow$ aggregate $slice[l_{a,slice}]$        ▷ aggregation
5: $buffer[l_{a,buffer}] \leftarrow events$                        ▷ read events
6: send $partial\_result, buffer[l_{a,buffer}], event\_rate$ to root node
```

Deco$_{sync}$ partially aggregates events in the *local slice*. The *local buffer* is a set of $2 * Delta_{a,Gi}$ events after the local slice. We formally define the local slice and local buffer as follows. Let $l_{a,slice}$ be the local slice size and let $l_{a,buffer}$ be the local buffer size of the node a. Equations 3 and 4 define the local slice size and buffer size as follows:

$$l_{a,slice} = \begin{cases} \hat{l}_{a,Gi} - \Delta_{a,Gi} & if\ \hat{l}_{a,Gi} > \Delta_{a,Gi}, \\ 0 & else, \end{cases} \tag{3}$$

$$l_{a,buffer} = 2 * \Delta_{a,Gi}. \tag{4}$$

Example: In the above numerical example, $l_{a,slice}$ is 0.6 million and $l_{a,buffer}$ is 2000.

Events in a local slice belong to either the current or the next local window. The prediction of the local window size is under- or over-estimated when the event rate changes. This may happen as the root node predicts the local window size before knowing the actual event rates. Deco$_{sync}$'s delta intercepts this up to a degree of $\Delta_{a,Gi}$. Hereby creating a local slice is a synchronous computation between all nodes. It is only created when the previous global window ends to make sure that the current local slice has the correct starting event and no events from the previous window are included in the current local slice. Deco$_{sync}$ performs incremental aggregation to all events in the local slice and outputs partial results to the root once the local slice ends. Deco$_{sync}$ creates the local buffer to buffer events and transmits all of these events directly to the root node when the buffer is full. We compute the local buffer size of global window i as follows. In some cases, there are late events and we will talk about this case in the later text (4.3.1).

The predicted local window size is larger than the local slice size but smaller than the sum of local slice size and local buffer size, i.e., $l_{a,slice} + l_{a,buffer}$. This is because the local node involves more events to its window to adapt to changing event rates. Once the local buffer is full, the local node sends the partial result of the local slice, the event rate, and the events in local buffer to the root node. The updates are propagated to the root node and Deco$_{sync}$ moves to the verification step.

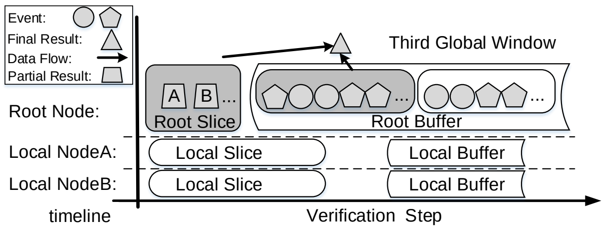


Figure 5: Verification step in the root node of Deco$_{sync}$

Verification. In the verification step, we verify the predicted local window sizes and calculate the final results. While the root node performs the verification local nodes wait for the confirmation. The root node first calculates the actual local window size for all local nodes based on the event rates sent by the local nodes. We let $\hat{l}_{a,Gi}$ be the predicted local window size and $l_{a,Gi}$ be the actual local window size. The prediction of node a is acceptable when $l_{a,Gi}$ conforms Equation 5 or 6, otherwise the prediction is wrong and we call this a prediction error:

$$l_{a,Gi} < \hat{l}_{a,Gi} + \Delta_{a,Gi}, \tag{5}$$

$$l_{a,Gi} >= \hat{l}_{a,Gi} - \Delta_{a,Gi}. \tag{6}$$

Here is an example of the root node performing the verification step (Algorithm 3). *Example.* In the previous example, let $l_{a,Gi}$ be 0.6005 million, the $\hat{l}_{a,Gi} - \Delta_{a,Gi}$ is 0.6 million and $\hat{l}_{a,Gi} + \Delta_{a,Gi}$ is 0.602 million. Thus, the prediction of node a is correct.

Algorithm 3 Deco$_{sync}$ performs verification step of window i on the root node.

```
Input: $partial\_result, buffer[l_{buffer}], event\_rate$
Output: $result$
1: $result \leftarrow$ aggregate $partial\_result$                         ▷ aggregation
2: $root\_buffer[] \leftarrow buffer[l_{buffer}]$                     ▷ read local buffers
3: $result \leftarrow$ aggregate $root\_buffer[l_{global} - l_{root,slice}]$     ▷ aggregation
4: if $\forall a : [l_{a,Gi} < \hat{l}_{a,Gi} + \Delta_{a,Gi}\ \&\ l_{a,Gi} >= \hat{l}_{a,Gi} - \Delta_{a,Gi}]$ then
5:     start next global window                               ▷ prediction correct
6:     return $result$
7: else
8:     start correction step                                  ▷ prediction wrong
9: end if
```

Next, we discuss the prediction error. $\hat{l}_{a,Gi} + \Delta_{a,Gi}$ is equal to $l_{a,slice} + l_{a,buffer}$. Local nodes only send events that are in local buffers and assume all events after local buffers belong to the next window. When $l_{a,Gi}$ is larger than $\hat{l}_{a,Gi} + \Delta_{a,Gi}$, i.e., Equation 5 is violated, the root node cannot collect all events of the current global window, so the result is incorrect. Furthermore, $\hat{l}_{a,Gi} - \Delta_{a,Gi}$ is equal to $l_{a,slice}$. When $l_{a,Gi}$ is smaller than $l_{a,slice}$, i.e., Equation 6 is violated, the partial result calculated by the local node is wrong and the root node cannot produce the correct result as well. Additionally, some cases also can result in prediction errors, and we will discuss these cases and how to fix prediction errors in the later text (Section 4.3).

When the prediction is correct, the root node starts to calculate the final result. It creates a global window that has two parts, root slice and root buffer, which are analogously defined as local nodes. Let n be the number of nodes in a decentralized network and $l_{root,slice}$ is the size of a root slice. The root slice size of the global window i is equal to the sum of local slice sizes of all local nodes. Formally, we compute the total root slice size as follows:

$$l_{root,slice} = \sum_{j=1}^{n} l_{j,slice}. \tag{7}$$

The root buffer size $l_{root,buffer}$ is computed as follows:

$$l_{root,buffer} = \sum_{j=1}^{n} l_{j,buffer}. \tag{8}$$

Example: The $l_{a,slice}$ and $l_{a,buffer}$ of node a are 0.6 million and 2000. Let $l_{b,slice}$ and $l_{b,buffer}$ of node b be 0.398 million and 2000. So root slice size is 0.998 and root buffer size is 4000.

As illustrated in Figure 5, Deco$_{\text{sync}}$ first creates a root slice and a root buffer on the root node and collects partial results from local node a and b and the statistics including the number of events and the first and the last event's timestamps. All partial results are put into the root slice and aggregated incrementally. Once the root slice has partial results from all local nodes, the root node ends the root slice. Also, all events sent from the local buffers are stored in the root buffer. Deco$_{\text{sync}}$ sorts the events in the root buffer by their timestamps and terminates the root buffer once there are $l_{root,buffer}$ events. Then Deco$_{\text{sync}}$ selects first $l_{global} - l_{root,slice}$ (i.e., 2000 as global window size is 1 million) events from the root buffer and aggregates them with the result of the root slice.

The root node calculates the final result and the actual local window sizes. The current global window ends, the root node then starts the next global window and repeats the prediction step. In the local nodes, the next window starts when new predicated local window sizes (e.g., local slice sizes and local buffer sizes) arrive. The local nodes first drop all events in the previous local window as they already contributed to an emitted result and then repeat the calculation step.

In addition, when the local window sizes of node a of two consecutive global windows are roughly the same $\Delta_{a,Gi}$ will be close to zero as Equation 2 shows. In this case, the predicted local window size stays the same and the local buffer size is almost zero. The local nodes only send partial results to the root node rather than additional raw events. Without sending raw events, even slight changes in event rates may result in incorrect results. Therefore, we record $\Delta_{a,Gi}$ for every global window and compute the average $\overline{\Delta}_{a,Gi}$ of the last m global windows. In this case, $\Delta_{a,Gi}$ is affected by multiple past windows. The parameter m is selected by the user and defines how aggressive Deco$_{\text{sync}}$ adapts to event rate changes. When m is large, the $\Delta_{a,Gi}$ is steady and changes slowly. In contrast, when m is small $\Delta_{a,Gi}$ is easily affected by changes in the event rate.

4.2.3 *Asynchronous scheme: Deco$_{async}$.*

Deco$_{\text{sync}}$ is a synchronous approach with three steps to process every global window. In the prediction step, local nodes are blocked and await information from the root node. In the calculation step, the root node waits for partial results and information before starting the verification step and continuing the prediction step of the next global window. To unblock local and root nodes, we propose Deco$_{\text{async}}$. Deco$_{\text{async}}$ has at least one communication flow between the root node and the local nodes. The first three global windows are processed similarly to Deco$_{\text{sync}}$. From the fourth global window, Deco$_{\text{async}}$ only has a calculation and a verification step for each window. Instead of waiting for new information from the root node, local nodes predict slice sizes and buffer sizes locally.

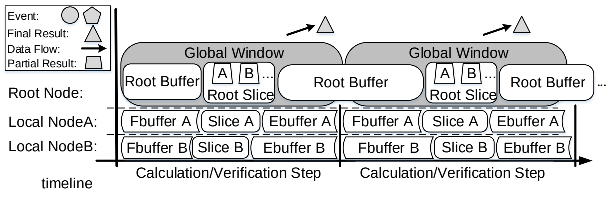


Figure 6: Calculation and verification steps of Deco$_{\text{async}}$

Calculation. In the calculation step, all local nodes have the predicted local window sizes and deltas from the previous global window. As illustrated in Figure 6, local nodes divide local windows into three parts, i.e., local front buffer (Fbuffer), local slice, and local end buffer (Ebuffer). We let $\hat{l}_a$ and Δ_a be the predicted local window size and delta of node a. In the following, we show the calculation step of Deco$_{\text{async}}$ in Algorithm 4.

Algorithm 4 Deco$_{\text{async}}$ perform calculation step of window i on the local node a.

```
Input: $\hat{l}_{a,Gi}$, $\Delta_{a,Gi}$: predicted window size and delta
Output: $partial\_result, Fbuffer, Ebuffer, event\_rate$
1: $l_{a,slice} = \hat{l}_{a,Gi} - 2 * \Delta_{a,Gi}$
2: $l_{a,Fbuffer} = l_{a,Ebuffer} = \Delta_{a,Gi}$
3: $slice[l_{a,slice}] \leftarrow events$                          ▷ read events
4: $partial\_result \leftarrow$ aggregate $slice[l_{a,slice}]$      ▷ aggregation
5: $Ebuffer[l_{a,Ebuffer}] \leftarrow events$                      ▷ read events
6: $Fbuffer[l_{a,Fbuffer}] \leftarrow events$                      ▷ read events
7: send $partial\_result, Fbuffer, Ebuffer, event\_rate$ to root node
```

We define the local slice size in Equation 9:

$$l_{a,slice} = \begin{cases} \hat{l}_a - 2 * \Delta_a & if\ \hat{l}_a > 2 * \Delta_a, \\ 0 & else. \end{cases} \tag{9}$$

The predicted local windows may not be the same as the actual local window. To make room for prediction error, we create an Fbuffer and an Ebuffer for every predicted local window. The Fbuffer is before the local slice and the Ebuffer is after the local slice. Let $l_{a,Fbuffer}$ and $l_{a,Ebuffer}$ be the Fbuffer and Ebuffer size of local node a, we define the Fbuffer and Ebuffer in Equation 10:

$$l_{a,Fbuffer} = l_{a,Ebuffer} = \Delta_{a,Gi}. \tag{10}$$

Example: Let $\hat{l}_a$ and Δ_a be 0.601 million and 1000. Then, $l_{a,slice}$ is 0.599 million, and $l_{a,Fbuffer}$ and $l_{a,Ebuffer}$ are 1000.

If $l_{a,slice}$ is 0, we calculate $l_{a,Fbuffer}$ and $l_{a,Ebuffer}$ as $\frac{\hat{l}_a}{2}$, otherwise they are calculated by Equation 10. The local nodes aggregate local slices and send partial results while all events in Fbuffers and Ebuffers are transmitted to the root node directly. The local nodes then start the calculation step of the next global window immediately without waiting for the message from the root node. In the next global window, the local nodes reuse the Fbuffer sizes, local slice sizes, and Ebuffer sizes of the previous global windows. In this case, the local nodes are not blocked and are able to process windows consecutively as the approximate solution but provide the correct results. Also, $\hat{l}_a$ and Δ_a will be updated once the local node receives the new local window size and delta. Otherwise, the local node reuses the same values from the previous calculation step.

Verification. In the verification step, the root node splits global windows into two parts, i.e., root slice and root buffer (see Figure 6). The root slice aggregates partial results from local slices and its size is defined below:

$$l_{root,slice} = \sum_{j=1}^{n} l_{j,slice}. \tag{11}$$

Example: Based on the above numerical example, the local slice size of node a and b are 0.599 million and 0.397 million. Thus, root slice size ($l_{root,slice}$) is 0.996 million. We present Algorithm 5 on the verification step.

Algorithm 5 Deco$_{\text{async}}$ perform verification step of window i on the root node.

```
Input: $partial\_result, Fbuffer, Ebuffer, event\_rate$
Output: $\hat{l}_{a,Gi}, \Delta_{a,Gi}, result$: predicted local window size and delta
 1: $result \leftarrow$ aggregate $partial\_result$                              ▷ aggregation
 2: $previous\_root\_buffer[] \leftarrow Fbuffer[l_{buffer}]$          ▷ read local Fbuffers
 3: $current\_root\_buffer[] \leftarrow Ebuffer[l_{buffer}]$           ▷ read local Ebuffers
 4: $result \leftarrow$ aggregate $previous\_root\_buffer[]$                   ▷ aggregation
 5: $result \leftarrow$ aggregate $current\_root\_buffer[l_{global} - l_{root,slice} - l_{root,buffer}]$   ▷ aggregation
 6: if $[l_{global} < l_{root,buffer} + l_{root,slice} + \hat{l}_{root,buffer}$ & $l_{global} >= l_{root,buffer} + l_{root,slice}]$ then
 7:     start next global window                                  ▷ prediction correct
 8:     $\hat{l}_{a,Gi} = l_{a,Gi-1}$
 9:     $\Delta_{a,Gi} = |l_{a,Gi-1} - l_{a,Gi-2}|$
10:     send $\hat{l}_{a,Gi}$ and $\Delta_{a,Gi}$ to local nodes
11:     return $result$
12: else
13:     start correction step                                     ▷ prediction wrong
14: end if
```

The root node has two root buffers, i.e., the previous root buffer and the current root buffer. The previous root buffer is from the previous global window and it is not empty if the previous window does not aggregate all events from it. Once the final result of the previous global window is calculated, all events belonging to the previous global window are removed from the previous root buffer. All events that are not removed are considered part of the current global window. The previous root buffer collects Fbuffer from local nodes. We let $l_{root,buffer}$ be the previous root buffer size and it is defined in Equation 12:

$$l_{root,buffer} = l_{root,buffer} + \sum_{j=1}^{n} l_{j,Fbuffer}. \tag{12}$$

Example: The previous root buffer size initially is 100 and the Fbuffer size of a and b are 1000 as well. The $l_{root,buffer}$ is 2100. Also, the root node creates a new root buffer for every global window, which is the current root buffer. All Ebuffer are put into the current root buffer. Let current root buffer size be $\hat{l}_{root,buffer}$ (Equation 12).

$$\hat{l}_{root,buffer} = \sum_{j=1}^{n} l_{j,Ebuffer}. \tag{13}$$

Example: As $l_{a,Ebuffer}$ and $l_{b,Ebuffer}$ are 1000, $\hat{l}_{root,buffer}$ is equal to 2000.

When the root node collects all partial results, Fbuffers, and Ebuffers, it starts to verify the local nodes' predictions. We let l_{global} be the global window size. The prediction is acceptable when l_{global} conforms Equation 14 and 15, otherwise the prediction is wrong and we perform the correction step.

$$l_{global} >= l_{root,buffer} + l_{root,slice}, \tag{14}$$

$$l_{global} < l_{root,buffer} + l_{root,slice} + \hat{l}_{root,buffer}. \tag{15}$$

Example: The l_{global} is 1 million, $l_{root,buffer} + l_{root,slice}$ is 0.9981 million, and $l_{root,buffer} + l_{root,slice} + \hat{l}_{root,buffer}$ is 1.0001 million. Thus, the predictions are correct.

When the prediction is correct, the root selects first $l_{global} - l_{root,slice} - l_{root,buffer}$ events from the current root buffer and aggregates those events to output the partial results. This partial result is aggregated with the root slice and previous root buffer to produce the final results. All events calculated in the current root buffer are dropped with the previous root buffer and root slice. Afterwards, the root node calculates the predicted local window sizes as well as deltas assigns them to the local nodes, and then starts the next global window. In this case, Deco$_{\text{async}}$ is able to process windows continuously the same as the approximate solution as the root node needs to wait for messages from local nodes while local nodes do not.

4.3 Correctness of Deco Approaches

Deco$_{\text{mon}}$ monitors the event rates of all local nodes and sends the actual local window sizes to local nodes. In this case, Deco$_{\text{mon}}$ always produces correct results even if the event rates change significantly. Deco$_{\text{sync}}$ and Deco$_{\text{async}}$ predict the local window sizes for the next global window and their results are verified in the verification step. The verification step ensures that any prediction errors can be detected. When the predictions are wrong, Deco$_{\text{sync}}$ and Deco$_{\text{async}}$ have to perform the correction step.

4.3.1 Correction Step of Deco$_{sync}$. if Deco$_{\text{sync}}$ predicts local window sizes correctly during the prediction step, the final results will be correct as well. However, in real decentralized networks, the event rates may change frequently and significantly. We, therefore, perform verification step to detect whether the prediction is correct or not. When the actual local window size conforms Equation 5, 6, the prediction is correct, otherwise there is a prediction error.

There are two scenarios that result in inaccurate predictions. In the first scenario, the event rate of a local node increases and the actual local window is larger than the predicted local window. To deal with this case, we create a slice and a buffer on the local node and the sizes of slice and buffer are larger than the size of predicted window sizes, i.e., slice size and buffer size are $l_{a,slice} + l_{a,buffer}$. The local nodes send more events than the predicted local window to the root node so that Deco$_{\text{sync}}$ can make room for the prediction. The root node determines which events from buffers belong to the current global window. In the second scenario, the event rate decreases and the actual local window is smaller than the predicted local window. To deal with this case, the size of local slice is smaller than the predicted local window, i.e., slice size is $l_{a,slice}$. The local node uses fewer events when calculating the partial results in order to make room for the prediction as well. The root node aggregates partial results and events from buffer to calculate correct results.

Event rates might change significantly and increase too much in a local node. The local slice and local buffer on that local node are quickly filled. After sending the message to the root node, the local node is blocked and waits for a message from the verification step before starting the next window. Therefore, the local node only needs to store one window in memory. Deco$_{\text{sync}}$ buffers all events in the memory.

In rare cases, the actual local window size ($l_{a,Gi}$) is larger than local slice and local buffer sizes ($l_{a,slice} + l_{a,buffer}$) and the prediction is wrong, i.e., predicted too less. That violate the Equation 5. Analogously, when the event rates decrease too much, the actual local window size ($l_{a,Gi}$) is smaller than slice size ($l_{a,slice}$), i.e., predict too much. The partial results sent from local nodes involve events belonging to the next global window and cannot be aggregated to the current global window, which violates Equation 6. Also, given that the global window size is fixed, an increased local window size of one node leads to a decrease of other node's local window size. Two cases usually happen in the same global window. To deal with these prediction errors, we perform correction steps to produce the correct results. The correction step is similar to the Deco$_{\text{mon}}$ as they use the actual local window sizes to calculate results.

Here, we discuss how Deco$_{\text{sync}}$ performs the correction step. As the root node has to calculate the actual local window size for all local nodes, it needs to wait for all local nodes to send event rates. Deco$_{\text{sync}}$ terminates the verification step only if the root node has received event rates from all local nodes. Firstly, the root node directly assigns all the actual local window sizes to the local nodes and informs them there is a prediction error as the actual local window sizes of all local nodes are calculated in the verification step. Secondly, once local nodes receive actual local window sizes and messages, they create local windows based on the actual local window sizes and calculate partial results. The local nodes then send partial results as well as the last event to the root node. This is because the actual window sizes are calculated from the event rates and they might not be the integers. The last event may or may not belong to the global window. Finally, the root node aggregates all partial results and last events that belong to the current global window and starts to process for the next global window. In the correction step, there are two communication flows, one from the root to inform local nodes and another one from local nodes to send correct partial results.

Deco$_{\text{sync}}$ provides support for backpressure mechanisms by introducing queues like Kafka [43]. Such mechanisms of queuing might introduce additional delays for decentralized processing of count windows as other nodes may have to wait until the queued events are in the SPE, which is the case in other known SPEs like Flink also. Still, event correctness is not compromised in Deco$_{\text{sync}}$ so we trade it against processing delay. Having guarantees for both correctness and low processing delay is an orthogonal direction and can be the future work of Deco.

4.3.2 *Correction Step of Deco$_{async}$.*

If the event rates change and Deco$_{\text{async}}$ cannot ensure that every prediction is correct, we employ a verification step to detect where the prediction is wrong. When the global window size violates either Equations 14 or 15, there is a prediction error. There are two scenarios that lead to prediction errors. In the root node, when the previous root buffer size and root slice size ($l_{root,buffer} + l_{root,slice}$) are larger than global window size, the root slices contain events that are not part of the current window. That violates the Equation 14, i.e., an overestimation. Additionally, when the previous root buffer size, root slice size, and current root buffer size, i.e., $l_{root,buffer}+l_{root,slice}+\hat{l}_{root,buffer}$, are smaller than global window size, the prediction violates the Equation 15. This is because the root node cannot collect all events of the current global window, i.e., predicted too less. To deal with these cases, Deco$_{\text{async}}$ performs correction step, which is similar to Deco$_{\text{mon}}$. Therefore, the correction step ensures Deco$_{\text{async}}$ produces correct results.

In the correction step, the root node sends actual local window size of the current window to local nodes and informs local nodes the prediction is wrong. The root node then collects partial results for the current window and calculates the final result. The root node also sends the predicted local window sizes and deltas for the next global window to local nodes. When local nodes receive a message from the root node, they update the newest predicted local window sizes, deltas, and watermarks. If the predictions are wrong, local nodes calculate correct results by the actual local window sizes and send partial results to the root node. As the local nodes are not blocked and they continue processing windows, all results after the wrong prediction are incorrect. Local nodes have to back to the window that has the wrong prediction and recalculate all local windows after that. If the predictions are correct, local nodes continuously process windows. Also, all events before a watermark are dropped since their windows are verified and predicted correctly.

Similar to Deco$_{\text{sync}}$, the memory requirements of Deco$_{\text{async}}$ are also bounded. Local nodes require memory to store events that are being processed. As soon as the window is processed and the results are verified and predicted correctly, the local node clears this window such that new events can be processed from datastream nodes. Similar to Deco$_{\text{sync}}$, Deco$_{\text{async}}$ also uses backpressure to deal with increasing event rates. This aligns with the backpressure mechanisms of current SPEs.

4.3.3 *Calculation of Event Rates.*

When the local buffer is full, the local node calculates the event rate and sends the event rate to the root node. The local node polls frequencies of data sources and calculates the event rate based on the frequencies. We assume that the interval is from the time of the last polling frequency to the time of the current polling frequency. So the interval of event rates is a bit larger than the lifespan of the actual local window. This is because, in general, the slice size and buffer size are larger than the actual local window size. Local nodes send more events than actual local windows to the root node. All events from the local buffer are ordered by timestamp and stored in the root buffer. The root node only selects the events it needs, so the results are still correct.

4.3.4 *Failure Model.*

Timeout and Watermark. We set timeouts for all local windows to deal with delayed events and missing messages. The timeout of local nodes can be provided by the users or calculated by the following equation. The root node's timeout is provided by users. For node a in global window Gi, let $\hat{l}_{a,Gi}$ and $\Delta_{a,Gi}$ be its predicted local window size and delta. Let f_a be the event rate. Its timeout $T_{a,Gi}$ is calculated as:

$$T_{a,Gi} = \frac{\hat{l}_{a,Gi} + 2 * \Delta_{a,Gi}}{f_a}. \tag{16}$$

When a local timeout is triggered, the local node sends events in the buffer and the new event rate to the root node. If the slice is still collecting events, the local node will terminate the slice and send its partial results to the root node. In this case, the root node still can receive correct partial results and events, since buffers always involve more events. Also, Deco selects the timestamp of the last event in the global window as the watermark. When starting a new global window the root sends the watermark to local nodes. Local nodes drop all events that have timestamps earlier than the watermark.

Node failures and unreliable networks. Deco supports crash failures of the root node and local nodes and can add and remove nodes during runtime. To add or remove a node, users have to inform the root node about the whereabouts of the local node. The root node then adds or removes the node from the Deco and sends the new topology to all other nodes. To handle failed local nodes, Deco utilizes a timeout to force local nodes to send information in time, which is similar to centralized solutions. Even if event rates decrease sharply the root node can still receive messages from the local nodes. When the root does not receive messages from one of the local nodes, it assumes the node has failed after the timeout. The root node then starts the correction step. If the local node permanently fails, the root node will elect a new local node that continues partial aggregation of the window. When the root node fails, Deco restarts the root node and asks local nodes to resend messages and then start processing. Additionally, the network might be unreliable and can drop or delay messages. When a delayed message is received, if it belongs to the previous global window, the message will be dropped, otherwise, the root node starts the correction step. In addition, Deco is also compatible with similar fault tolerance techniques used in traditional SPEs like Flink and will sport more failure models in future work.

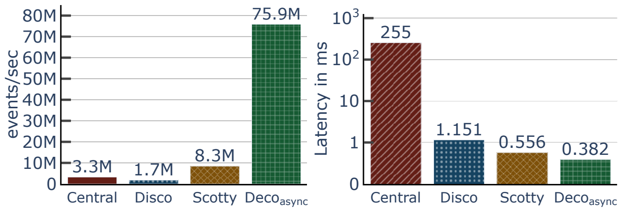
In decentralized networks, data sources are distributed on different nodes and have different setups. Deco will break when there is a data source always sending events with late timestamps. Current centralized aggregation solutions also meet the same issues. In this case, processing events with wrong timestamps results in a global window including events that should be in the next windows. To deal with this issue Deco can utilize existing solutions [17, 62] to synchronize the clocks of all data sources and local nodes, which is the same as the current solutions. Deco is also compatible with similar fault tolerance techniques used in traditional SPEs like Flink, e.g., heart-beats, storage checkpointing, state snapshots with the Chandy-Lamport algorithm, and rewinding the event stream in case of failures. Therefore, Deco is able to incorporate crash-stop, omission, and crash recovery. Deco does not support Byzantine failure models.

5 EVALUATION

In this section, we evaluate the performance of Deco and compare our system with state-of-the-art approaches.

Experimental Design. We conduct experiments on a 10-node cluster (unless specified otherwise) with 25 Gbit/s Ethernet connection. Each node has two 18-Core Intel Xeon Gold 5220S CPUs and 96 GB main memory. We run Deco on Ubuntu 20.04 and OpenJDK 1.8.0.312 64 bit edition. In our experiments, we measure throughput, network utilization, and latency.

Evaluation Metrics. We measure sustainable throughput. In this setup, the system processes incoming data without an ever-increasing backlog [38]. We also compute the sustainable network utilization of every single node in each system and then aggregate them. We measure latency with processing-time rather than event-time, which calculates the time from when the event arrives at the node to when the result or partial result involving the event is produced. For example, a local node starts to calculate latency when the event arrives and output latency when the partial result of this event is sent to the root node. We output the final latency by aggregating all latency provided from different nodes and the network overhead of latency is not considered. As data generators are deployed on local nodes, the event-time when an event is created is the same as the processing-time when an event arrives. We, therefore, avoid coordination omission [58] which would lead to significant underestimation of latency [26].



(a) End-to-end Throughput **(b) End-to-end Latency**

Figure 7: Performance of different approaches in comparison to Deco$_{\text{async}}$. Deco$_{\text{async}}$ outperforms centralized approaches by $10\times$ in throughput and $100\times$ in latency.

Evaluated Approaches. We compare to four approaches, including centralized aggregation: Central, Scotty [60], Disco [6], approximate decentralized aggregation (Approx) against our approaches[3] Deco$_{\text{mon}}$, Deco$_{\text{sync}}$, and Deco$_{\text{async}}$. All approaches are implemented in Java. All can perform incremental aggregation except for Central. Central is a straightforward approach that forwards all raw events to the root node and performs the window aggregation on the root node. Central is analog to an implementation of common SPEs like Flink and Spark [9, 66]. The Scotty baseline utilizes the Scotty API and shares partial results between concurrent windows to reduce memory usage and avoid duplicate processing of a single event. Scotty baseline processes events with the centralized aggregation. Disco is a decentralized system that uses the Scotty API, so it also can share partial results. However, Disco only performs decentralized aggregation for time-based windows and processes count-based windows with centralized aggregation. Compared to Scotty, Disco uses only one thread to receive, process, and send events. We modify the data input of Disco to use our data generator. Approx is introduced in Section 4 and it performs decentralized aggregation for count-based windows but outputs wrong results. This is because Approx cannot adapt the local window size when event rates change and thus cannot ensure correctness.

Data Generators. We simulate a decentralized network environment. The local nodes run data generator instances that simulate externally generated events. We simulate multiple parallel data streams by starting each stream with a different offset in the dataset. An event contains three elements, e.g., *id*, *value*, *timestamp*. The data generator gives every event a sequential *id* and a *timestamp*. We use the DEBS 2013 dataset [53] to generate *value*s. Our dataset was collected by a real-time locating system deployed on a soccer field. The local nodes replay the dataset from different positions so that we can simulate a real deployment. The data generator provides a parameter to define the event rate change, e.g., the event rate is 100 events/s, and it changes between 95 to 105 events/s if the parameter is 5%.

5.1 End to End Performance

We first compare Deco$_{\text{async}}$, Scotty, Disco, and Central. We focus on the throughput, network utilization, and latency metrics of each system.

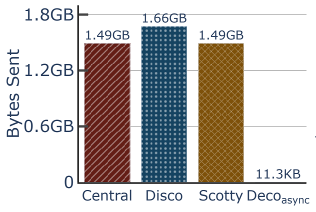
End-to-End Throughput and Latency. All approaches are executed on a 9-node cluster with one root and eightlocal nodes. We process a tumbling count-based window and the aggregation function *sum* with the event rate change of 1%. We then measure the performance of four approaches.

[3]https://github.com/hpides/Deco#deco

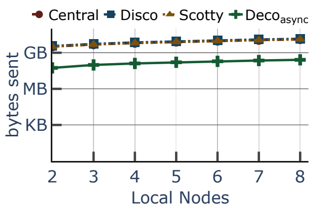
In Figure 7a, we evaluate a count-based window of size 1 million with eight local nodes. The throughput of Deco$_{async}$ is 75.9 million events/s while that of Scotty is 8.3 million events/s. The results confirm that decentralized aggregation of partial results has a positive effect on throughput, as we see an increase of almost 10×. The partial aggregation close to the source utilizes the additional processing power of the local nodes. This is clearly an advantage over centralized approaches like Disco and Central, which have lower throughputs of 1.7 million events/s and 3.3 million events/s, respectively. We also see that Central outperforms Disco as Disco only uses a single thread to receive, process, and send events, while Central and Scotty do not.

In Figure 7b, we show the latency (in ms) of each approach with eight local nodes. We can observe that Central has the highest latency because it collects all events to windows and executes aggregation functions individually for all events, once the window ends. The other approaches, instead, process events incrementally. Disco uses the Scotty API to process events but only uses a single thread, which also results in an increase in latency. While Scotty's approach uses separate threads to send, receive, and process events which is why it almost matches the performance of Deco$_{async}$ in latency.

In summary, Deco$_{async}$ outperforms all approaches with respect to throughput and is on par with other approaches in terms of latency.



(a) Local node data transfer

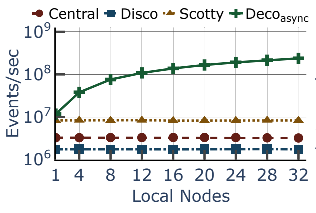


(b) Multi node data transfer

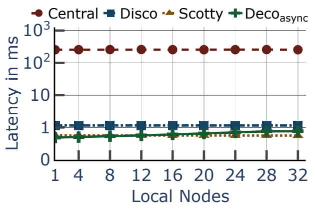
Figure 8: Network utilization of different approaches in comparison to Deco$_{async}$. Deco$_{async}$ saves network cost by up to 99% compared to centralized approaches.

Network Utilization. In this experiment, we evaluate the network utilization of the different approaches. All approaches compute the sum over a tumbling count-based window. Every local node receives 100 million events and the event rate change is 1%. We then add gradually more local nodes to the topology up to 8 nodes and calculate their network utilization.

In Figure 8a, we show the network utilization of all approaches in a 2-node cluster, one local and one root node. In comparison to the other approaches, Deco$_{async}$ utilizes only a fraction of the network resources. Different from the other approaches Deco$_{async}$ avoids sending raw events whenever possible, which saves 99% of the network resources. Our investigation showed that the network cost of Disco is higher than Central and Scotty because it uses strings to send events and messages. In the second experiment, we gradually increase the number of local nodes in each topology as seen in Figure 8b so that there are more events sent between the root and local nodes. Compared with the 2-node cluster, the network utilization of Deco$_{async}$ increases suddenly in the 3-node cluster. Local nodes have to transmit a fraction of events to make room for prediction. Additionally, we learn that the network costs of approaches increase linearly (notice the log scale of the y axis) since the overall number of events increases. Deco$_{async}$ still shows superior results.



(a) Local node throughput



(b) Local node latency

Figure 9: Scalability of different approaches in comparison to Deco$_{async}$. Deco$_{async}$ outperforms centralized approaches and its performance scales linearly.

In summary, Deco$_{async}$ can reduce network cost dramatically since it performs partial aggregations on local nodes and especially avoids sending raw events to the root node. In contrast, centralized approaches send all data from local nodes to the root node, which is why the total network cost increases linearly when the number of local nodes increases.

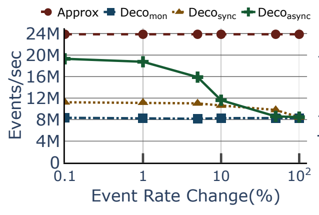
Scalability. Next, we investigate how our approach scales with additional local nodes. All approaches are initially working on a 2-node cluster with one root node and one local node. We gradually add more local nodes to the topology and measure the throughput and latency. All approaches compute the sum over a tumbling count-based window with an event rate change of 1%.

In Figure 9a, we gradually increase the number of local nodes from 1 to 32. To eliminate the effect of small size windows, we also gradually increase the window size (Section 5.2). All local nodes connect to the root node. We observe that the throughput of Deco$_{async}$ scales linearly and the other approaches stay the same because Deco$_{async}$ offloads computations from the root node to local nodes and aggregates partial results to produce the final results. We also notice the trend of a gradual slowdown in the throughput. This is because more events are sent and processed on the root node. Scotty and Disco can perform decentralized aggregation when processing time-based windows, but for count-based windows, they still perform centralized aggregation. Thus, they do not advantage of adding more local nodes to the setup. As Disco only uses a single thread to send, receive, and process events, it has less throughput than Scotty and Central. Scotty outperforms Central since it performs incremental aggregation but Central does not.

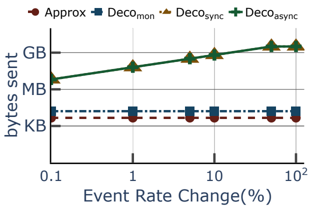
We measure latency in the same experiment as shown in Figure 9b. The latency of Deco$_{async}$ increases very slowly with the number of local nodes. More local nodes send events to the root node, which leads to more calculations on the root node. Scotty, Disco, and Central on the other side, send all raw events to the root node and perform a centralized aggregation and thus observe constant latency.

In summary, when processing count-based windows, decentralized aggregation benefits from additional local nodes and we see an increase in throughput. We also notice a gradual slowing trend in throughput growth and a slow increase in latency. Deco$_{async}$ has good scalability and can be extended over 32 nodes.

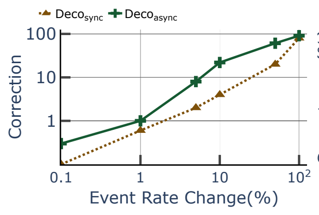
Microbenchmark. We also conduct a microbenchmark by removing the root node and conducting the aggregation solely based on local nodes. For this, we modify the Deco$_{mon}$ approach and denote this approach as Deco$_{monlocal}$ as follows. In the initialization step, local nodes communicate with each other to exchange event rates. The verification steps are moved to each local node. Only if a local node collects all event rates from other



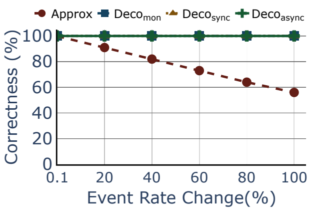
(a) Throughput / event rate



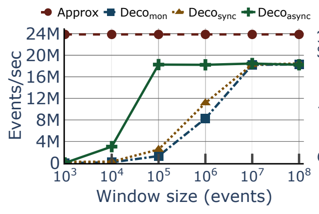
(b) Data transfer / event rate



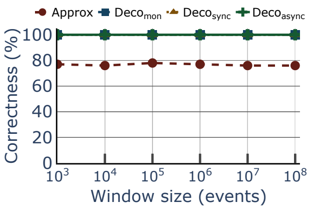
(c) Correction steps



(d) Correctness / event rate



(e) Throughput / window size



(f) Correctness / window size

Figure 10: Adaptivity of Deco in comparison to Approx

nodes, it start to calculate window sizes. The calculation step is the same as Deco$_{mon}$. The root node then has to inform local nodes to start the next window. There are still three flows for each window in Deco$_{monlocal}$. We use the same setup as before but 32 local nodes and one root node. We see that the latency of Deco$_{monlocal}$ (10.24 ms) is larger than Deco$_{mon}$ (0.526 ms). This is because of higher synchronization overhead, in each initialization step, every local node has to synchronize with others.

5.2 Adaptivity Performance

In this experiment, we measure the effects of event rate changes on the performance of the approaches Approx, Deco$_{mon}$, Deco$_{sync}$, and Deco$_{async}$. All approaches are deployed on a three-node cluster: two local nodes and a root node. We process a tumbling count-based window with a *sum* function.

Adaptivity to Event Rate. In this experiment, we gradually vary the event rate change range from 0.1% to 100%. In Figure 10a and Figure 10b we evaluate the throughput and network utilization. Here the Approx method has an optimal throughput since Approx is able to move all calculations from the root node to local nodes without transmitting events. While our approach Deco$_{async}$ roughly matches the throughput of Approx at about 20 million events/s when the event rates less than 1%. Our other approaches like Deco$_{mon}$ and Deco$_{sync}$ suffer from low throughput. This is because in the local node, Deco$_{async}$ starts a new window once the previous window ends, but Deco$_{mon}$ and Deco$_{sync}$ have to wait for new messages from the root. When the event rates change frequently, the throughput of Deco$_{async}$ drops and is even lower than Deco$_{sync}$ since Deco$_{async}$ can easily have prediction errors and lead to more correct steps. Also, Deco$_{async}$ reuses the predicted local window sizes and deltas to process the following local windows without waiting. Once the prediction is wrong, Deco$_{async}$ has to recalculate all windows after the wrong one, which affects throughput significantly. Additionally, Deco$_{mon}$ is roughly close to Approx which has the lowest network cost because both of them only transmit partial results and a few messages. The network costs of Deco$_{sync}$ and Deco$_{async}$ increase linearly when the event rate change increases because of higher communication costs. This is because the buffer size changes and the local node sends more events to the root.

Also, the root node has the most calculation load when event rates change frequently since most of the events are sent to the root node. In contrast, when event rates change slightly local nodes have the most calculation load. This is because the event rate change affects local buffer and slice sizes.

In the next experiment Figure 10c, we change the event rates and study how many correction steps are executed for every 100 count-based windows. We let every local node send or process 100 million events and evaluate each system. We see that Deco$_{async}$ executes more correct steps than Deco$_{sync}$. This is because Deco$_{sync}$ is a synchronous approach and it will block the global window and execute the correction step immediately when the prediction step is wrong. In this case, the wrong prediction will not affect the next global window. However, Deco$_{async}$ executes more correction steps since it is asynchronous and cannot fix the wrong prediction in time. When the change range is growing larger, Deco$_{async}$ and Deco$_{sync}$ have to perform more correction steps and in fact the same as Deco$_{mon}$.

We evaluate the correctness in Figure 10d. We use Central as the ground truth and compare every window of Central and other approaches to calculate how many events from other approaches are the same in the Central window. This way, we know how many events in fact belong to the current window, which means how many events are processed correctly. We then divide the total number of correctly processed events by the total number of events (100 million) to calculate correctness. Deco$_{mon}$, Deco$_{sync}$, and Deco$_{async}$ ensure 100% correctness which is the same as the centralized aggregation approach. When event rates change frequently the correctness of Approx decreases linearly, while other approaches remain correct.

In summary, when the event rate changes frequently, all approaches are correct except Approx. Deco$_{async}$ is most close to Approx (Optimal) with respect to throughput when event rates change slightly. Deco$_{sync}$ has better throughput when event rates change moderately. Also, the changing event rates lead to an increase in network costs which behaves linearly to the buffer size.

Adaptivity to Window Sizes. We evaluate all approaches with a tumbling count-based window and an event rate change of 1%. We vary the window size and measure the throughput.

Figure 10e shows that Deco is most beneficial when window sizes grow larger. This is intuitive because for larger window sizes we start seeing the benefit of decentralized aggregation, otherwise centralized aggregation is sufficient for smaller window sizes. Another interesting observation is that the benefits of Deco$_{async}$ are sooner visible than other approaches Deco$_{sync}$ and Deco$_{mon}$ due to the asynchronous (non-blocking) nature of communication in Deco$_{async}$. Also after seeing an increase up to 20 million events, the throughput is saturated and we might see a further increase when new local nodes are added. In Figure 10f, we evaluate correctness in an unstable setup. We set the event rate change as 50% and vary the window size. We see that Deco$_{mon}$, Deco$_{sync}$, and Deco$_{async}$ have 100% correctness when the window size changes.

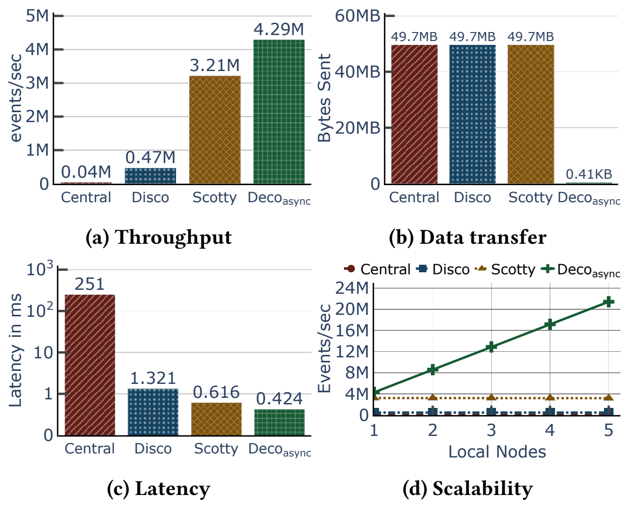


Figure 11: Performance of different approaches in comparison to Deco$_{async}$ in a Raspberry Pi cluster.

In summary, Deco$_{mon}$, Deco$_{sync}$, and Deco$_{async}$ suffer from the small window size since Deco$_{mon}$ and Deco$_{sync}$ are blocked and Deco$_{async}$ is busy with executing the correction step.

5.3 Performance on IoT nodes

We evaluate the performance of the approaches in a more realistic IoT setup with Raspberry Pis Model 4B as local nodes. They feature 1G Ethernet, a 4-Core Cortex-A72, 8 GB memory, with Debian 11 64-bit. We use one Intel node as the root (*Intel node*) and Raspberry Pis as the local nodes. We use a tumbling window of size one million and an event rate change of 1%.

Figure 11a shows that Deco$_{async}$ has a maximum throughput of 4.3 million events/s as it performs decentralized aggregation. In Figure 11b and Figure 11c, we see that Scotty, Disco, and Central transfer 49 MB per second, which is also the maximum bandwidth of the Raspberry Pis. Deco$_{async}$ has the lowest latency. Figure 11d shows that the throughput of Deco$_{async}$ scales linearly with the number of Raspberry Pis. Scotty's throughput stays at 3.2 million events/s even if we add more nodes. In summary, Deco$_{async}$ still shows the lowest latency and highest throughput when running on a realistic setup with less powerful devices.

6 RELATED WORK

Current SPEs such as Flink, Spark Streaming, etc. [9, 43, 56, 59, 65, 66] process large-scale high speed data streams. SPEs split the potentially unbound data streams into bounded windows with different window types, window measures, and aggregation functions. To efficiently process windows in decentralized networks, state-of-the-art solutions push down window aggregation close to the stream source. TAG [46, 47], Cougar [63], LEACH [29], Directed Diffusion [31], and Disco [6] assume a tree-structured network topology and only support time-based windows. For count-based windows previous approaches either centrally aggregate or re-partition streams [10, 14, 44, 60]. The decentralized approach Web Liquid Streams [2] for heterogeneous hardware infrastructures provides only approximate global window aggregation. To the best of our knowledge, no previous decentralized approach provides correct count-based window aggregation.

From the communication taxonomy perspective, Deco is a hierarchical approach that takes advantage of the decomposability of aggregation functions[4]. Hierarchic-based approaches [8, 16, 46, 49, 52] exploit the decomposability characteristic but do not support count-based windows. Some of these approaches apply approximate partial aggregation close to the source [49] which leads to incorrect results. Deco also pushes down work close to the source but maintains correct results. Averaging based approaches like [15, 16, 33–35, 40] share averaged results and Sketch-based approaches like [3, 21, 23, 24, 49, 54] share lightweight approximated data structures between nodes. Digests based approaches allow approximating more complex aggregation functions like median. Randomized approaches [4, 27, 30, 37, 41, 42, 48, 50] sample the data stream. Dropping events randomly reduces network transfer rates like in [57]. Different from these approaches Deco produces correct results while still reducing the number of events transferred via the network.

Other approaches tackle the problem of adapting to changing event rates. Dhalion [25] analyzes complex query pipelines and detects operators that are bottlenecks regarding hardware resources and operator characteristics. Based on this information event rates are regulated. Different from Dhalion Deco focuses on low-level adaption to changing event rates rather than high-level control of event rates. A complex approach by Bertsimas et al. [7] approximates event rate estimations based on linear optimization and machine learning. Deco achieves good performance with a much simpler technique that incorporates only at the previous event rate variance, more advanced predictions could also be applied in future work.

Approaches for elastically controlling distributed applications either minimize hardware utilization [5, 11, 45, 51] or resource costs [22]. They focus on decentralized allocation and deallocation of resources. In contrast to Deco, these approaches focus on high-level operator splitting and operator placement and miss opportunities of low-level optimizations.

7 CONCLUSION

In this paper, we present the decentralized aggregation approach Deco, which enables count-based windows in decentralized networks. Deco adapts to changing event rates and ingests events from decentralized streams. Deco pushes count-based window aggregation down to local nodes and predicts the local window sizes based on the input distribution such that the window aggregation can be performed locally. Our approach ensures the same correct results of count-based windows as the centralized aggregation solutions and processes windows in a decentralized fashion without the need for a central node to collect all events. Our evaluation shows that Deco reduces network traffic by up to 99% and scales linearly with the number of nodes.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (ref. 414984028), the European Unions Horizon 2020 research and innovation programme (ref. 957407), the HPI research school on Data Science and Engineering, as well as DFKI Darmstadt.

REFERENCES

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

[4]We use the classifications of [36]

[2] Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso. 2015. Decentralized stream processing over web-enabled devices. In *Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings 4*. Springer, 3–18.

[3] Carlos Baquero, Paulo Sérgio Almeida, and Raquel Menezes. 2009. Fast estimation of aggregates in unstructured networks. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*. IEEE, 88–93.

[4] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. 2003. *Estimating aggregates on a peer-to-peer network*. Technical Report. Technical report, Stanford University.

[5] Mehdi Mokhtar Belkhiria and Cédric Tedeschi. 2020. A fully decentralized autoscaling algorithm for stream processing applications. In *Euro-Par 2019: Parallel Processing Workshops: Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, Revised Selected Papers 25*. Springer, 42–53.

[6] Lawrence Benson, Philipp M Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. 2020. Disco: Efficient Distributed Window Aggregation.. In *EDBT*, Vol. 20. 423–426.

[7] Dimitris Bertsimas and Vassilis Digalakis. 2021. Frequency estimation in data streams: Learning the optimal hashing scheme. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[8] Yitzhak Birk, Idit Keidar, Liran Liss, Assaf Schuster, and Ran Wolff. 2006. Veracity radius: capturing the locality of distributed computations. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. 102–111.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[10] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1201–1210.

[11] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems* 87 (2018), 171–185.

[12] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 725–736.

[13] Luca Catarinucci, Danilo De Donno, Luca Mainetti, Luca Palano, Luigi Patrono, Maria Laura Stefanizzi, and Luciano Tarricone. 2015. An IoT-aware architecture for smart healthcare systems. *IEEE internet of things journal* 2, 6 (2015), 515–526.

[14] Huan Chen, Yijie Wang, Yuan Wang, and Xingkong Ma. 2016. GDSW: a general framework for distributed sliding window over data streams. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 729–736.

[15] Jen-Yeu Chen, Gopal Pandurangan, and Dongyan Xu. 2006. Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. *IEEE Transactions on Parallel and Distributed Systems* 17, 9 (2006), 987–1000.

[16] Laukik Chitnis, Alin Dobra, and Sanjay Ranka. 2008. Aggregation methods for large-scale sensor networks. *ACM Transactions on Sensor Networks (TOSN)* 4, 2 (2008), 1–36.

[17] Bong Jun Choi, Hao Liang, Xuemin Shen, and Weihua Zhuang. 2011. DCS: Distributed asynchronous clock synchronization in delay tolerant networks. *IEEE Transactions on Parallel and Distributed Systems* 23, 3 (2011), 491–504.

[18] Cisco. 2021. *Cisco Catalyst 2960-X and 2960-XR Series Switches Data Sheet*. Cisco. Retrieved Nov 5, 2021 from https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-2960-x-series-switches/datasheet_c78-728232.html

[19] Cisco. 2023. *Cisco Catalyst Wireless Gateway CG113 Data Sheet Data*. Cisco. Retrieved Dec 5, 2023 from https://www.cisco.com/c/en/us/products/collateral/routers/catalyst-wireless-gateway-cg110-series/nb-06-cat-wireless-gateway-cg113-ds-cte-en.html?oid=dstswt029984

[20] Louis Columbus. 2016. *Internet of Things-number of connected devices worldwide 2015-2025*. IHS. Retrieved Nov 26, 2020 from https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[21] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. 2004. Approximate aggregation techniques for sensor databases. In *Proceedings. 20th International Conference on Data Engineering*. IEEE, 449–460.

[22] Scott Eisele, Michael Wilbur, Taha Eghtesad, Kevin Silvergold, Fred Eisele, Ayan Mukhopadhyay, Aron Laszka, and Abhishek Dubey. 2022. Decentralized Computation Market for Stream Processing Applications. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 36–46.

[23] Yao-Chung Fan and Arbee LP Chen. 2008. Efficient and robust sensor data aggregation using linear counting sketches. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[24] Yao-Chung Fan and Arbee LP Chen. 2010. Efficient and robust schemes for sensor data aggregation based on linear counting. *IEEE Transactions on Parallel and Distributed Systems* 21, 11 (2010), 1675–1691.

[25] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[26] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. 2017. Coordinated omission in nosql database benchmarking. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017).

[27] Ayalvadi J Ganesh, A-M Kermarrec, Erwan Le Merrer, and Laurent Massoulié. 2007. Peer counting and sampling in overlay networks based on random walks. *Distributed Computing* 20, 4 (2007), 267.

[28] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1 (1997), 29–53.

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

[30] Keren Horowitz and Dahlia Malkhi. 2003. Estimating network size from local information. *Inform. Process. Lett.* 88, 5 (2003), 237–243.

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

[32] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316.

[33] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. 2005. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)* 23, 3 (2005), 219–252.

[34] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2009. Fault-tolerant aggregation by flow updating. In *Distributed Applications and Interoperable Systems: 9th IFIP WG 6.1 International Conference, DAIS 2009, Lisbon, Portugal, June 9-11, 2009. Proceedings 9*. Springer, 73–86.

[35] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2010. Fault-tolerant aggregation for dynamic networks. In *2010 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 37–43.

[36] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2014. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 381–404.

[37] Tomasz Jurdziński, Mirosław Kutyłowski, and Jan Zatopiański. 2002. Energy-efficient size approximation of radio networks with no collision detection. In *Computing and Combinatorics: 8th Annual International Conference, COCOON 2002 Singapore, August 15–17, 2002 Proceedings 8*. Springer, 279–289.

[38] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518.

[39] Sean Dieter Tebje Kelly, Nagender Kumar Suryadevara, and Subhas Chandra Mukhopadhyay. 2013. Towards the implementation of IoT for environmental condition monitoring in homes. *IEEE sensors journal* 13, 10 (2013), 3846–3853.

[40] David Kempe, Alin Dobra, and Johannes Gehrke. 2003. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 482–491.

[41] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Ken Birman, and Alan Demers. 2005. Decentralized schemes for size estimation in large and dynamic groups. In *Fourth IEEE International Symposium on Network Computing and Applications*. IEEE, 41–48.

[42] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Kenneth P Birman, and Alan J Demers. 2007. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. *Journal of Systems and Software* 80, 10 (2007), 1639–1658.

[43] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Athens, Greece, 1–7.

[44] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *Acm Sigmod Record* 34, 1 (2005), 39–44.

[45] Zhen Liu, Ao Tang, Cathy H Xia, and Li Zhang. 2009. A decentralized control mechanism for stream processing networks. *Annals of Operations Research* 170 (2009), 161–182.

[46] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2002. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*.

[47] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2003. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 491–502.

[48] Sandeep Mane, Sandeep Mopuru, Kriti Mehra, and Jaideep Srivastava. 2005. Network size estimation in a peer-to-peer network. (2005).

[49] Amit Manjhi, Suman Nath, and Phillip B Gibbons. 2005. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 287–298.

[50] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. 2006. Peer counting and sampling in overlay networks: random walk methods. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. 123–132.

[51] Timo Michelsen. 2014. Data stream processing in dynamic and decentralized peer-to-peer networks. In *Proceedings of the 2014 SIGMOD PhD symposium*. 1–5.

[52] Shinji Motegi, Kiyohito Yoshihara, and Hiroki Horiuchi. 2006. DAG based in-network aggregation for sensor network monitoring. In *international Symposium on Applications and the Internet (SAINT'06)*. IEEE, 8–pp.

[53] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 grand challenge. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*. 289–294.

[54] Rudi Poepsel-Lemaitre, Martin Kiefer, Joscha Von Hein, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. In the land of data streams where synopses are missing, one framework to bring them all. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1818–1831.

[55] Srinivasa Prasanna and Srinivasa Rao. 2012. An overview of wireless sensor networks applications and security. *International Journal of Soft Computing and Engineering (IJSCE)* 2, 2 (2012), 2231–2307.

[56] Matthias J Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. 1–10.

[57] Yufei Tao, Xiaocheng Hu, and Miao Qiao. 2017. Stream sampling over windows with worst-case optimality and ℓ-overlap independence. *The VLDB Journal* 26, 4 (2017), 493–510.

[58] Gil Tene. 2016. *How NOT to Measure Latency*. IHS. Retrieved Mar 26, 2016 from https://www.infoq.com/presentations/latency-response-time/

[59] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[60] Jonas Traub, Philipp Marian Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2021. Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–46.

[61] Ioan Ungurean, Nicoleta-Cristina Gaitan, and Vasile Gheorghita Gaitan. 2014. An IoT architecture for things from industrial environment. In *2014 10th International Conference on Communications (COMM)*. IEEE, 1–4.

[62] Yik-Chung Wu, Qasim Chaudhari, and Erchin Serpedin. 2010. Clock synchronization of wireless sensor networks. *IEEE Signal Processing Magazine* 28, 1 (2010), 124–138.

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

[64] Wang Yue, Lawrence Benson, and Tilmann Rabl. 2023. Desis: Efficient Window Aggregation in Decentralized Networks. *26th International Conference on Extending Database Technology (EDBT 2023)* (2023).

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[66] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 423–438.

[67] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2019. The NebulaStream Platform: Data and application management for the internet of things. *arXiv preprint arXiv:1910.07867* (2019).