Efficient Stream Processing in Decentralized Networks

Wang Yue
Supervised by Prof. Tilmann Rabl
Hasso Plattner Institute, Potsdam, Germany
wang.yue@hpi.de

ABSTRACT

Internet-of-things (IoT) devices are widely used in industry as well as in research and are deployed in many applications. These massive amounts of devices are connected in large decentralized networks and produce unbounded data streams with continuous data. To process these data streams timely, current stream processing engines (SPEs) collect all data in a centralized data center. This approach leads to high network utilization and can create a bottleneck in the data center, as all data is transmitted via the network and results are computed centrally. State-of-the-art solutions push down partial window aggregations to machines that are near data streams. However, these solutions are limited to a single simple query. In this paper, we present our work on three solutions for different decentralized aggregations: Desis, Deco, and Dema, which significantly improve the performance of stream processing in decentralized networks. Our solutions reduce network traffic by up to 99.9 %.

VLDB Workshop Reference Format:
Wang Yue. Efficient Stream Processing in Decentralized Networks. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

The Internet of Things (IoT) is pervasive in many domains such as healthcare, Industry 4.0, and smart cities [8]. These applications involve massive amounts of IoT devices distributed across many decentralized networks. To process the unbounded data streams generated by these devices, current stream processing engines (SPEs) such as Apache Flink [2] and Apache Spark Streaming [13] split data streams into windows. Once a window ends, they perform window aggregation and output the results. Current Stream Processing Engines (SPEs) perform centralized window aggregation to process data from decentralized networks. In this approach, both the creation of windows and the processing of computations occur at a central node. All data is transmitted via the internet to this central node and the center is the only node that processes data. To reduce network overhead and resource consumption, state-of-the-art approaches [1, 6, 14] suggest decentralized aggregation. This method shifts window aggregations to devices that are closer to the data streams. These devices create windows, perform partial window aggregations, and output partial results. Instead of sending all data events to the central node, only the partial results are transmitted.

There are three issues in current stream processing: (i) decentralized window slicing, (ii) decentralized aggregation of count-based windows, and (iii) decentralized aggregation of windows with non-decomposable functions.

Decentralized window slicing. The processing of data streams often involves many simultaneous queries, resulting in concurrent windows. These windows might overlap and lead to redundant computations and unnecessary resource consumption if processed individually. Current approaches [3, 10] use window slicing to merge the partial results of slices instead of repeatedly calculating overlapping parts of windows. However, these approaches are limited to sharing partial results only between windows with the same aggregation functions and are not suitable for decentralized setups.

Aggregation of count-based windows. These approaches are limited to time-based windows with decomposable functions. Time-based windows can be easily divided based on time intervals, allowing multiple nodes to 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, making them challenging to split. This is because the center node requires prior information on the incoming event rates from local nodes to ensure the correctness of splitting windows.

Non-decomposable functions. Windows with decomposable functions, e.g., sum and count, can calculate final results by merging partial results from split windows. Non-decomposable functions, e.g., median and quantile, require sorting all data and cannot be accurately computed using partial results only.

To deal with the above limitations we propose three approaches. (1) We design *Desis* [12], a stream processing system that efficiently processes concurrent windows in decentralized networks. (2) We propose *Deco* [11] that has three schemes and enables processing count-based with decentralized aggregation while output correct results. (3) We introduce *Dema* that pushes down windows with non-decomposable functions close to data sources to calculate window aggregations decentralized.

In this paper, we will discuss these three approaches which are already published (Desis, Deco) or currently work in progress (Dema). Our work aims to propose a stream processing system that can efficiently process concurrent windows with different window types and aggregation functions in decentralized networks.

2 DESIS

Desis is a stream processing system designed to process multi-queries in decentralized networks. In this section, we discuss how to handle concurrent windows with decentralized aggregation.

2.1 System Overview

In a decentralized topology, there are many local nodes and intermediate nodes but only one root node. Local nodes connect to

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

data streams and link to the root node via intermediate nodes. In complex networks, multiple intermediate nodes interconnect local and root nodes. In simple networks, local nodes may connect to the root node directly, and intermediate nodes might not be necessary.

2.2 Aggregation Engine

Desis proposes an aggregation engine to share partial results between multiple windows with different window types and different aggregation functions. Before discussing our approach, we define two terms: (i) Query-group: A set of queries that can share partial results between them. (ii) Punctuation: Used to mark the lifespan of a window, the start punctuation (*sp*) denotes the window's beginning, and the end punctuation (*ep*) indicates its end.

Also, we propose aggregate operators as the most basic aggregation functions. We support many operators, e.g., *count*, *sum*, *multiplication*, *square root*, *decomposable sort*, and *non-decomposable sort*. Instead of directly processing aggregation functions, the aggregation engine breaks down them into operators. Given two overlapping windows, one calculates sum and the other one calculates average, the aggregation engine shares the sum operator between them. is because the average function can be decomposed into sum and count operators

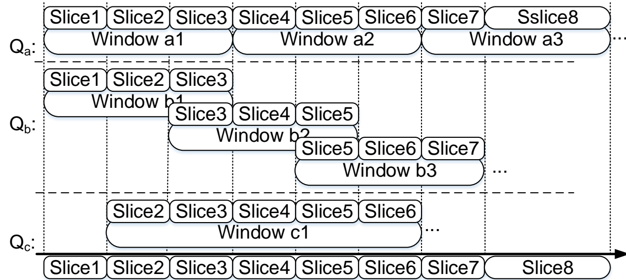


Figure 1: Aggregation engine processing multiple queries

In Figure 1, we process three example queries. Q_a has tumbling windows with a average aggregation, Q_b has sliding windows and a sum function. Query Q_c contains session windows and a count function. As these queries can share partial results in between, they are in the same query-group. The aggregation engine splits three functions into two operators (sum and count) and then processes queries. Whenever there is a punctuation, e.g., *sp* or *ep*, the aggregation engine terminates the current slice and creates a new slice. For example, window *a1* (Q_a) has three slices: *slice1*, *slice2*, and *slice3*. The *slice1* is shared between window *a1* and window *b1*. The *slice2* is shared between window *a1*, window *b1*, and window *c1*. The *slice3* is shared between window *a1*, window *b1*, window *b2*, and window *c1*. Instead of calculating three aggregation functions, the system only processes two operators for each slice. The result of window *b1* is merged from partial results of *slice1*, *slice2*, and *slice3*.

2.3 Decentralized Aggregation

Instead of pushing down windows to local nodes, Desis slices windows on the root and sends slices. Desis can perform window slicing and share partial results on all nodes in the decentralized networks. Also, Desis does not send partial results per window but per slice.

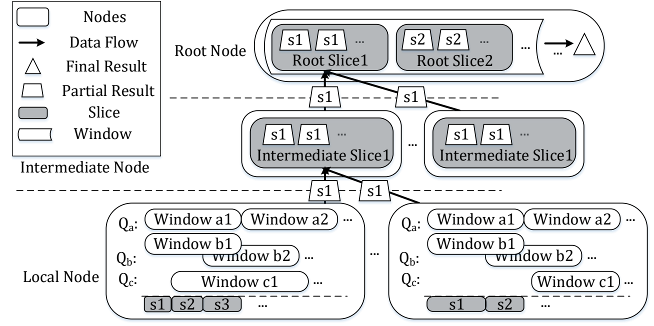
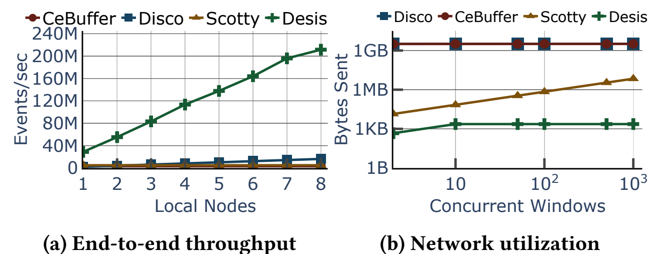


Figure 2: Window aggregation in decentralized networks

In Figure 2, the three queries are the same from the Figure 1 but they are time-based. On the root node, all three queries (Q_a, Q_b, and Q_b) are grouped into the same query-group, which consists of two operators: *sum* and *count*. Then, the window attributes of this query-group are distributed to local and intermediate nodes. On the local node, the aggregation engine slices windows into local slices and performs incremental aggregation for every incoming event. When a slice ends, the aggregation engine sends the partial results to the intermediate node. On the intermediate node, the aggregation engine creates the intermediate slice to collect partial results from local nodes connected to it. Once the intermediate slice has all partial results from local nodes, it aggregates results again and sends the aggregated partial results to the root node. For example, the intermediate *s1* only collects two local *s1*s. On the root node, the partial results of intermediate slices are incrementally aggregated into root slices. For example, root *s1* is terminated only if it has collected partial results from all intermediate *s1*. The local node also marks the partial result with the *sp* or *ep* of its slice. When there is an *ep*, the root node aggregates the window and outputs the final result.

2.4 Evaluation

We evaluate Desis with three baselines: Cebuffer (hand-coded), Disco [1], and Scotty [10]. Cebuffer and Scotty are centralized approaches, Scotty uses window slicing techniques but Cebuffer does not. Disco is a decentralized solution that can only share partial results on local nodes and is limited to windows with the same aggregation functions.



(a) End-to-end throughput (b) Network utilization

Figure 3: End-to-end throughput and network utilization

In Figure 3a, we run all systems initially on one local node and gradually add new local nodes, and they process 1000 concurrent windows. In Figure 3b, all systems are deployed on two local nodes and gradually scale to process more concurrent windows. We learn

that Desis significantly outperforms all baselines and scales well. Also, Desis can reduce over 99 % network cost.

3 DECO

Deco is a decentralized aggregation approach that enables the split of count-based windows in decentralized networks. In this section, we discuss technical details about Deco.

3.1 System Overview

Deco has local nodes, intermediate nodes, and the node root node in its setup, but intermediate nodes only transfer data. We define the number of events received per second as *event rate*. We let the f be the event rate and l be window size. For a decentralized network with n local nodes, the total event rate of the root node is $f_{root} = \sum_{i=1}^{n} f_i$. Also, 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.

An approximate solution is to calculate the local window sizes for each local node once and then keep them fixed. The local nodes reuse these same local window sizes to perform window aggregation regardless of changes in the event rate. Because this method relies on static local window sizes, it does not produce correct results.

3.2 Deco$_{\text{mon}}$

To adapt to changing event rates and produce correct results, we propose Deco$_{\text{mon}}$. It monitors the event rates and updates local window sizes for every global window. Deco$_{\text{mon}}$ has three steps: (i) Initialization: All local nodes send event rates to the root node. (ii) Verification: The root node calculates local window sizes and assigns them to local nodes. (iii) Calculation: Local nodes create windows based on local window sizes and send partial results to the root node. The root node aggregates these partial results to produce the final results. Although Deco$_{\text{mon}}$ moves computations from the root node to local nodes, it requires three communication rounds between local nodes and the root node for each global window.

3.3 Deco$_{\text{sync}}$

To reduce communication overhead, we propose Deco$_{\text{sync}}$ a synchronized approach that uses predicted local window sizes instead of actual local window sizes. Deco$_{\text{sync}}$ has three steps: prediction, calculation, and verification. In the first two global windows Deco$_{\text{sync}}$ is the same as the Deco$_{\text{mon}}$.

Prediction. In Deco$_{\text{sync}}$, we assume that the event rates change slightly and the local window sizes of two consecutive global windows are close. Given a node a, We let the predicted local window size be $\hat{l}_{a,Gi}$, which is the actual local window size of the previous window. To correct potential prediction errors, we define Δ, the difference between the local window sizes of the previous two consecutive windows.

Calculation. Once the local node (a) receives the predicted local window size and Δ, it creates a new local window. The local window is divided into two parts, local slice and local buffer. Local slice size is equal to $\hat{l}_{a,Gi}$ minus Δ and local buffer size is equal to $2 * \Delta_{a,Gi}$. Deco$_{\text{sync}}$ aggregates all events in the local slices and sends partial results to the root node. Also, Deco$_{\text{sync}}$ transmits all events in the buffer and the event rate to the root node.

Verification. The root node verifies the predicted local window sizes with event rates from local nodes. The prediction of node a is acceptable when $l_{a,Gi}$ conforms Equations (1) and (2), otherwise the prediction is wrong and we call this a prediction error.

$$l_{a,Gi} < \hat{l}_{a,Gi} + \Delta_{a,Gi} \tag{1}$$

$$l_{a,Gi} >= \hat{l}_{a,Gi} - \Delta_{a,Gi} \tag{2}$$

If the prediction is correct, the root node collects partial results to the root slice and local buffers to the root buffer. It aggregates the root slice with events from the root buffer that belong to the current window to output the final results. The root node then starts the prediction step of the next window. There are still two communication rounds for every global window.

3.4 Deco$_{\text{async}}$

To further reduce communication overhead, we design Deco$_{\text{async}}$ an asynchronous approach that moves the prediction from the root to local nodes. There are only two steps: calculation and verification.

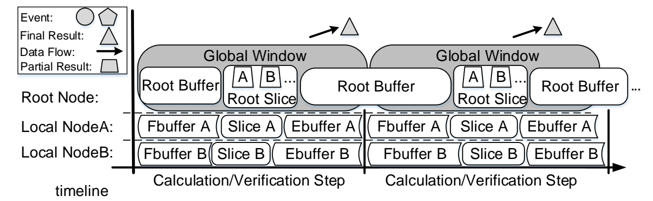


Figure 4: Calculation and verification steps of Deco$_{\text{async}}$

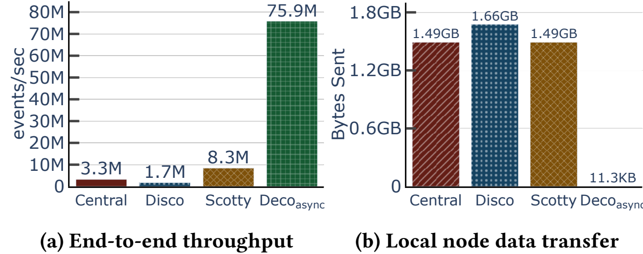
In the calculation step, local nodes predict the local window sizes and Δ from the previous windows. In Figure 4, the local window is divided into three parts. Local nodes send events from Fbuffer and Ebuffer, partial results of local slices, and event rates to the root node. They then immediately begin the calculation step of the next global window without waiting for the message from the root node. In the verification step, the root node verifies the predictions made by the local nodes. If the predictions are correct, the root node aggregates the root slice with events from root buffers. The root node then outputs the results and updates the information of the local nodes. In this case, the root node and local nodes do not need to wait for each other, resulting in only one communication round.

3.5 Correctness of Deco Approaches

Deco$_{\text{mon}}$ monitors the event rates and always produces correct results even if the event rates change significantly. Deco$_{\text{sync}}$ and Deco$_{\text{async}}$ predict window sizes and their predictions are verified in the verification step. If the prediction is wrong, the system will ask for the correction step, which is similar to the verification and calculation steps of Deco$_{\text{mon}}$. Therefore, both Deco$_{\text{sync}}$ and Deco$_{\text{async}}$ always produce correct results.

3.6 Evaluation

We compare $\text{Deco}_{\text{async}}$ with three baselines: Central, Disco, and Scotty. The central is the same as Cebuffer.



(a) End-to-end throughput **(b) Local node data transfer**

Figure 5: End-to-end throughput and network utilization

In Figure 5a, all systems are processed on eight local nodes. While in Figure 5b, the cluster only has two local nodes. The results show that $\text{Deco}_{\text{async}}$ has the highest throughput and lowest network cost since it moves calculations from the root node to local nodes and only sends partial results and a few events via the internet.

4 DEMA

Dema is a decentralized approach designed to process windows with non-decomposable functions in decentralized networks. Dema currently only supports functions that require sorting, such as median and quantile. The core concept of Dema is similar to Deco, which involves multiple steps of coordination between local nodes and the root node. In Figure 6, Dema has three steps: prediction, correction, and calculation. We illustrate this process using a time-based tumbling window with the median function as an example.

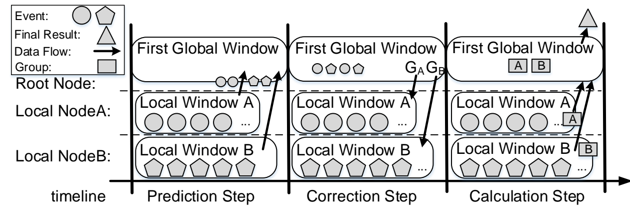


Figure 6: Three steps of Dema

In the prediction step, the local node creates a window and collects data. Once the window ends, the local node sorts events and sends candidate events to the root node. We define Δ as the prediction rate, it can be configured by users. For example, if Δ is set to 5, the local node divides the window into groups of size 5. The local node selects the first event of each group and sends the selected events and range of each group to the root node.

In the correction step, the root node sorts groups and removes all independent groups at the beginning and end. As the root node knows the range of each group, it can calculate which group includes the median. The root node then requests the local node to send all events belonging to that group. Also, the groups come from different local nodes, and they may overlap with each other. In this case, the median is involved with multiple groups from different local nodes. In the calculation steps, the root node receives all the required events and computes the median value.

5 RELATED WORK

Current work on window slicing [4, 9] can share partial results between windows, but is limited to tumbling and sliding windows. The centralized approaches Scotty [10] support arbitrary time-based windows, but windows with different functions are still processed individually. Disco [1] employs decentralized window aggregation but it only executes window slicing on local nodes. Existing efforts to split count-based windows or windows with non-decomposable functions [5, 7] are based on approximated aggregation or execute sampling on data streams, which leads to incorrect results.

6 CONCLUSION

In this paper, we present three approaches to improve window aggregations in decentralized setups. We present Desis, a stream processing system that can efficiently process multiple queries in decentralized networks. Desis facilitates the sharing of partial results among overlapping windows with varying window types and aggregation functions. We also describe the Deco approaches, including three schemes, Deco_{mon}, $\text{Deco}_{\text{sync}}$, and $\text{Deco}_{\text{async}}$, which split count-based window aggregation in decentralized networks. Deco adapts to changing event rates and ensures correct results. Additionally, we introduce Dema, which allows decentralized aggregation to support non-decomposable functions.

REFERENCES

[1] Lawrence Benson et al. 2020. Disco: Efficient Distributed Window Aggregation. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, Vol. 20. 423–426.

[2] 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).

[3] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-hoc Shared Stream Processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 607–622.

[4] Jin Li et al. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34, 1 (2005), 39–44.

[5] Amit Manjhi, Suman Nath, and Phillip B Gibbons. 2005. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 287–298.

[6] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. 2021. VergeDB: A Database for IoT Analytics on Edge Devices.. In *CIDR*.

[7] Rudi Poepsel-Lemaitre et al. 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.

[8] Yuya Sasaki. 2021. A survey on IoT big data analytic systems: Current and future. *IEEE Internet of Things Journal* 9, 2 (2021), 1024–1036.

[9] Georgios Theodorakis et al. 2020. LightSaber: Efficient window aggregation on multi-core processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2505–2521.

[10] Jonas Traub et al. 2021. Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems. *Transactions on Database Systems (TODS)* 46, 1 (2021), 1–46.

[11] Wang Yue et al. 2024. Deco: Fast and Accurate Decentralized Aggregation of Count-Based Windows in Large-Scale IoT Applications. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 412–425.

[12] Wang Yue, Lawrence Benson, and Tilmann Rabl. 2023. Desis: Efficient Window Aggregation in Decentralized Networks. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 618–631.

[13] Matei Zaharia et al. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SIGOPS Symposium on Operating Systems Principles, SOSP*. ACM, 423–438.

[14] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020*.