Disco: Efficient Distributed Window Aggregation

Lawrence Benson[1] Philipp M. Grulich[2] Steffen Zeuch[2,3] Volker Markl[2,3] Tilmann Rabl[1]
[1]Hasso Plattner Institute, University of Potsdam [2]Technische Universität Berlin [3]DFKI GmbH
{lawrence.benson,tilmann.rabl}@hpi.de,{grulich,steffen.zeuch,volker.markl}@tu-berlin.de

ABSTRACT

Many business applications benefit from fast analysis of online data streams. Modern stream processing engines (SPEs) provide complex window types and user-defined aggregation functions to analyze streams. While SPEs run in central data centers, wireless sensors networks (WSNs) perform distributed aggregations close to the data sources, which is beneficial especially in modern IoT setups. However, WSNs support only basic aggregations and windows. To bridge the gap between complex central aggregations and simple distributed analysis, we propose Disco, a distributed complex window aggregation approach. Disco processes complex window types on multiple independent nodes while efficiently aggregating incoming data streams. Our evaluation shows that Disco's throughput scales linearly with the number of nodes and that Disco already outperforms a centralized solution in a two-node setup. Furthermore, Disco reduces the network cost significantly compared to the centralized approach. Disco's tree-like topology handles thousands of nodes per level and scales to support future data-intensive streaming applications.

1 INTRODUCTION

Modern business use-cases often require the analysis of high-volume data streams. To efficiently process such large amounts of data, stream processing engines (SPEs) provide complex windows and aggregations. But to perform these complex aggregations, state-of-the-art SPEs such as Apache Flink [2], Apache Spark Streaming [16], and Apache Storm [12] require the data to be collected in a single data center. Current research approaches to improve the performance of window aggregation, such as Scotty [13, 14], Cutty [3], and Pairs [7] also require data to be centrally available. However, efficiently analyzing an ever-increasing data volume requires streams to be processed on multiple nodes as the central collection of data quickly becomes a limiting factor in processing latency and network cost [17]. While SPEs require data to be available centrally, wireless sensor networks (WSN) perform aggregations on multiple nodes close to the data sources. WSNs drastically reduce the network costs by actively leveraging the distribution of incoming data streams. However, previous work on WSNs, such as TAG [10] or Cougar [15], provides only simple aggregation operations on basic window types.

To bridge the gap between complex central aggregation in SPEs and distributed basic aggregation in WSNs, we propose Disco, a distributed complex window aggregation approach. While SPEs processes incoming data in one central stream, Disco leverages the distribution of incoming data streams to perform distributed window creation and aggregations closer to the sources. We propose multiple strategies to efficiently merge distributed windows and their respective aggregates. With this approach, we benefit from both the reduced network cost of WSNs and the complex analysis of SPEs.

In this paper, we make the following contributions: 1.) We propose Disco, an approach for distributed complex window aggregation that does not require raw data collection on single nodes. 2.) We introduce window merging strategies to distribute the processing of common window types while maintaining correct aggregation semantics. 3.) We evaluate Disco and show that the throughput of our approach scales linearly with the number of nodes as well reduces the network cost drastically compared to state-of-the-art central window aggregation techniques.

The rest of the paper is structured as follows. In Section 2, we present the foundations of window aggregation that Disco is built upon. In Section 3, we present our distributed aggregation approach for arbitrary window types and aggregation functions. Before concluding, we present our evaluation of these distributed aggregation concepts implemented in our prototype in Section 4.

2 BACKGROUND

In this section, we present the background of distributed window aggregation. We first present the two window types on which we build, as well as two classes of aggregation functions. We then introduce stream slicing for efficient window aggregations.

Window Types. For this work, we distinguish between two types of windows, *context-free* and *context-aware* windows [9, 14]. In this case, context refers to the state that is required to calculate the start- and end-bounds of all the windows up to time t.

Context-free (CF) windows do not require any state and their bounds can be computed statically. For a time t, all window bounds can be computed without processing a single event. Examples of CF windows are tumbling and sliding windows [1, 9].

Context-aware (CA) windows require some kind of state to determine the window bounds, i.e., the windowing function needs to see either previous or future events to know when a window ends. An example of a CA window is a session window [1]. The windowing function needs to see future events after t in order to know that a session has terminated at t.

Aggregation Functions. For distributed aggregation, we distinguish between two classes of aggregation functions, *decomposable* and *holistic* (or *non-decomposable*) [4, 5]. Decomposable functions are aggregations for which the aggregation can be performed on subsets of the data and merged afterwards, e.g., $\texttt{sum}\{1, 2, 3, 4\} = \texttt{sum}\{1, 2\} + \texttt{sum}\{3, 4\}$. We call the aggregation values on subsets of the data *partial aggregates*. Other examples of decomposable functions are `avg`, `count`, and `max`.

Holistic functions are all functions that are not decomposable, i.e., they require all values to perform the correct aggregation. There is no partial aggregation state for these functions. Examples of holistic functions are `median` and `quantiles`.

Stream Slicing. In Disco, we apply general stream slicing [14] to efficiently aggregate windows. Stream slicing divides windows into non-overlapping slices. The slices then store either *i)* a running, partial aggregate or *ii)* the individual events, depending on the aggregation function. The final aggregation result is computed from the individual slices that fall in the window's range. For commutative, decomposable aggregations only partial aggregates are needed, leading to a large memory reduction [14].

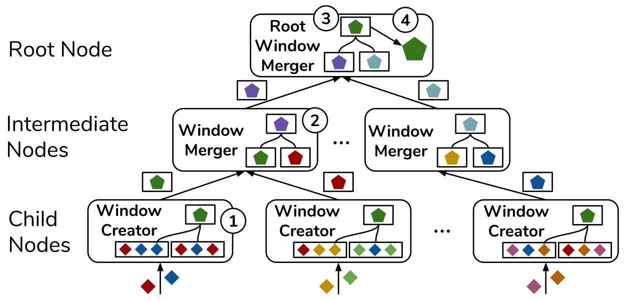


Figure 1: Disco's Architecture.

Each event belongs to exactly one slice, thus avoiding redundant storage and computation compared to other windowing techniques such as *Buckets* [9]. Also, slices represent logical event groups that can be transferred between nodes more efficiently than individual events. We refer to previous work [8, 14] for a detailed description of stream slicing and its advantages.

3 THE DISCO APPROACH

Efficiently aggregating windowed streaming data is a core task for modern SPEs. State-of-the-art window aggregation techniques require data to be available centrally on a single node. We introduce Disco, an approach for distributed window aggregation, which overcomes the central collection of raw events. Disco processes the incoming streams independently of each other and creates independent windows and aggregations on multiple nodes. These windowed aggregations are then merged to produce the final aggregation result. Disco's components communicate in a tree-like structure, in which leaf nodes (called *child nodes* in Disco) create independent windows and inner nodes (called *intermediate nodes*) merge them. This independence allows Disco to scale by adding new child nodes to process more events and by adding intermediate nodes to process more child nodes.

In the remainder of this section, we provide an architecture overview of Disco (Section 3.1) followed by a detailed description of Disco's window merging strategies (Section 3.2) and its distributed computation of aggregation functions (Section 3.3).

3.1 Architecture

We show Disco's architecture in Figure 1. Disco consists of three main components, the *root node*, the *intermediate nodes*, and the *child nodes*. These individual components communicate in a tree-like structure, where each node, except the root, communicates with exactly one parent node and each node can have many children, limited only by its network and processing power. Each event stream connects to one child node, which in turn receives the raw events. The *Window Creator* on each child node creates slices and windows according to user-specified queries ①. The child nodes then pass on partial aggregates for each window to their parent. On the intermediate nodes, a *Window Merger* then merges incoming windows according to the strategies presented in Section 3.2 ②. On the root node, the *Root Window Merger* finally merges all partial windows ③ and performs the final aggregation to retrieve the result for a given window ④. As the nodes process events and windows independently, there can be an arbitrary number of child nodes and intermediate levels, depending on the scale that the system requires.

Our implementation relies on the Scotty library[1] [14] for the window and slice creation step ①. Because Scotty requires data to be available centrally, we extend its windowing concepts to support distributed windows.

[1]https://github.com/TU-Berlin-DIMA/scotty-window-processor

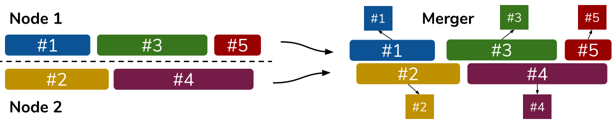


Figure 2: No merging for unique windows.

3.2 Distributed Window Merging

Depending on the window characteristics (i.e., type and measure), Disco selects the appropriate window creation and merging strategy. In particular, Disco differentiates between the two window types: *context-free* and *context-aware* (as described in Section 2). Disco processes incoming event streams and resulting windows independently on child nodes. In order to create the correct *global* window result over all streams, Disco merges the individual windows. To this end, Disco has to determine which partial windows belong to the same global window. We present three merging strategies for distributed window aggregation in Disco, *i)* unique window merging, *ii)* context-free merging, and *iii)* context-aware merging. For our explanation, we assume there are k independent child nodes producing windows and one *Root Window Merger* that merges the windows to produce a global result.

Window Merging Operations. In order to perform distributed aggregations, we use the three operations: *lift*, *combine*, and *lower* to model our aggregation functions [11]. *lift* converts a single input value x to an aggregation into a partial aggregate, e.g., $x = 5 \longrightarrow \langle$sum: 5, count: 1$\rangle$ for `avg`. *combine* merges two partial aggregates into a new partial aggregate, e.g., $\langle 5, 1\rangle \oplus \langle 7, 2\rangle \longrightarrow \langle 12, 3\rangle$. *lower* converts the partial aggregate to the final result, e.g., $\langle 12, 3\rangle \longrightarrow 12/3 = 4$.

To merge multiple windows, we extend the *slice-merge* operation [14] for windows. The *window-merge* operation $\uplus$ takes two windows w_1 and w_2 and creates a new window w with a merged aggregate state ($w.state$ = `combine`$(w_1.state, w_2.state)$), as well as the according bounds ($w.start = min(w_1.start, w_2.start)) \wedge w.end = max(w_1.end, w_2.end)$).

Unique Window Merging. The first merging strategy that Disco uses is based on *unique* windows. A unique window is a window for which data is present in only one stream. With no matching data, there are no further windows to merge the unique window with. This strategy is applicable to queries where there are no overlapping keys on different nodes and the query is defined on individual keys only. We show this in Figure 2. A Window Merger that receives a unique window w, emits w unchanged and the Root Window Merger calls `lower`$(w.state)$ to retrieve the final aggregation result. For unique windows, we benefit from a distributed aggregation compared to central processing, as we do not need to transfer raw data to the root node and we distribute the aggregation cost across all child nodes. An example for a unique window is a smart home setting in which we calculate the average temperature per house.

Context-Free Window Merging. Disco determines to which global window a partial context-free window belongs by its start and end bounds. As the bounds are statically computed and require no context, they are identical on all nodes. Thus, equivalent partial windows have equal start and end bounds. We show this for a sliding window on two nodes in Figure 3. For each global window w with $w.start = x$ and $w.end = y$, we collect the k matching partial windows out of all windows W, $\{w_i \in W \mid w_i.start = x \wedge w_i.end = y\}$ and merge them into a

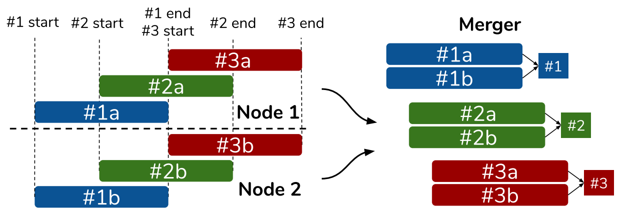


Figure 3: Merging distributed context-free windows.

new global result window $w = w_1 \uplus w_2 \uplus \ldots \uplus w_k$. The global result is then calculated by calling lower($w.state$) in the *Root Window Merger*. An example of a context-free window is a smart home setting in which we calculate the average temperature of all houses for the last hour.

Context-Aware Window Merging. Unlike context-free windows, the creation of context-aware windows cannot generally be distributed. As CA windows require some form of context, Disco cannot make assumptions about whether this context needs to be viewed centrally or can be viewed independently. If the window requires a global view of the context, we cannot create windows independently on multiple nodes. An example of such a global view is a count-based window, which uses a global event count to compute the window bounds.

However, certain CA windows, such as session windows, can benefit from a distributed aggregation. Session windows terminate if the event stream contains a period of inactivity (gap). In Disco, we define a period of inactivity across all distributed streams as a *global window gap*. A timestamp t is *active* in the set of all windows W if $\exists w \in W : w.start < t \land w.end > t$. A timestamp t is thus *inactive* if it is not active, i.e., there are no windows spanning across t. A period of inactivity is bound by two timestamps t_{start} and t_{end} between which there are no active timestamps. Disco leverages this knowledge to merge session windows in a distributed manner. Session windows are created independently on child nodes and the *Window Merger* then checks if there is an overlap between received windows. If an incoming window w_{in} overlaps with an existing window w_{ex}, it is merged to produce a new window $w_{new} = w_{ex} \cup w_{in}$ in their place. Two windows overlap if, and only if $w_{old}.start \leq w_{new}.end + gap \land w_{new}.start \leq w_{old}.end + gap$. For session windows, an overlap between two windows needs to take the session gap into account as events within the gap would cause the session to continue in a global stream. An arriving window can merge multiple windows, e.g., if it is a very long window or it closes the gap between two currently non-overlapping windows. We show this in Figure 4, where window #1b combines the windows #1a and #1c because it overlaps with both of them. An example for a context-aware window is a smart home setting in which we calculate the average temperature of certain houses as long as they are actively heating.

3.3 Distributed Window Aggregation

While the window type generally decides whether a window is computed centrally or distributed, the class of aggregation function determines which data needs to be transferred between nodes. For decomposable functions (e.g., `sum`, `avg`, `count`), Disco transfers partial aggregate values. The partial aggregates are then merged by the window merger.

As holistic functions require all data for a correct aggregation, Disco needs to transfer all events to the root. However, Disco does not send individual events but *slices*. For holistic functions, the slices contain the raw events and Disco uses these slices to send logical event groups between nodes. As each slice is immutable and uniquely identifiable, Disco sends it only once, regardless of how many windows it appears in. As all holistic functions require central processing, the root node is the only node that needs to store slices to performs aggregations on the events. Intermediate nodes do not process the data in the slices and thus, do not need to store the slices. Intermediate nodes only keep track of which slices they have sent to avoid duplicate transmission.

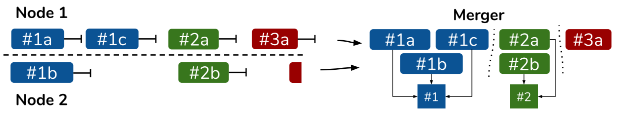


Figure 4: Merging distributed context-aware windows.

In summary, local and context-free windows can generally be distributed, while context-aware windows require certain data characteristics for distribution. If a window or aggregation function requires a global ordering of events (e.g., count-based windows), Disco cannot distribute the window creation across multiple nodes but requires central processing. For holistic aggregations, Disco transfers slices between nodes and for decomposable functions, it sends only partial aggregate values.

4 EVALUATION

In this section, we experimentally evaluate Disco's scalability (Section 4.1) and its network efficiency (Section 4.2). We choose `avg` and `median` as representative decomposable respectively holistic aggregation functions, as they show similar characteristics to other functions of the same group. We execute our experiments on a cluster consisting of 20 nodes. Each node has an AMD Opteron 6128 @ 2.0 GHz with 16 physical cores and 32 GB of RAM. All nodes are running Ubuntu 18.04.3 LTS, OpenJDK 12.0.2 64 bit, and are connected via Gigabit LAN. Furthermore, Disco and our experiments are available on GitHub[2].

4.1 Scalability

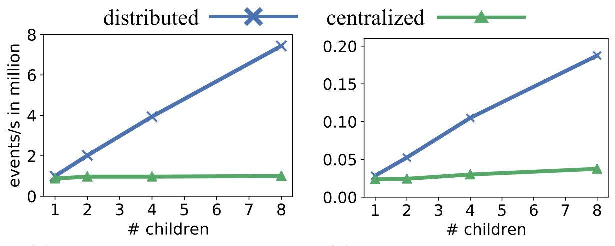
In this experiment, we compare the scalability of Disco's distributed window aggregations to a centralized implementation.

Workload. We evaluate the throughput of a one-second tumbling window query for a decomposable as well as a holistic aggregate function. We define throughput as sustainable if the system can handle incoming events without an ever-increasing backlog at the sender [6]. In the centralized implementation, the child nodes forward the raw events without processing.

Results. In Figure 5, we observe that Disco scales nearly linearly with the number of child nodes for both aggregation types. For decomposable aggregation functions (e.g., `avg` in Figure 5a), each child node can process around one million events per second. In the centralized approach, the root node becomes the bottleneck, as it processes all raw data centrally. As a consequence, it is limited by the single node performance (~1 million events/s). In contrast, Disco's root node receives only one partial window per second per child node instead of all raw events. Thus, Disco allows for linear scaling as the majority of the window aggregations is processed independently across all child nodes.

Furthermore, Disco provides scalability for holistic window aggregation functions (e.g., `median` in Figure 5b). Overall, the `median` aggregation performs significantly worse than the `avg` aggregation as it requires the centralized accumulation of all

[2]https://github.com/hpides/disco



(a) avg aggregate function (b) `median` aggregate function

Figure 5: Scalability of different aggregations in Disco.

events at the root node. As a consequence, the centralized approach is limited to a throughput of 0.025 million events/s. In contrast, Disco also performs window creation and merging of holistic states on multiple nodes in parallel. Thus, the root node only receives up to eight windows per second in this experiment. Consequently, Disco is able to scale nearly linearly for the holistic `median` function as the final `median` calculation on the root does not become a bottleneck for Disco.

Further scalability experiments show that the root node can process thousands of holistic and tens of thousands of decomposable windows per second before it becomes a bottleneck. Thus, Disco scales to support thousands of child nodes.

Summary. In this experiment, we showed that Disco scales linearly with the number of nodes for both decomposable as well as holistic aggregation functions. Even for functions that require central aggregation but can be windowed independently, Disco outperform centralized approaches significantly.

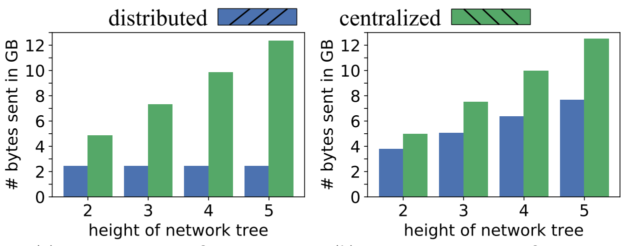
4.2 Network Cost

In this experiment, we evaluate the overall network costs of distributed and centralized aggregations.

Workload. We scale the height of the network topology to evaluate the network impact of the number of hops between child nodes and the root. We measure the total bytes sent for a one-second tumbling window query on 100 million events.

Results. Overall, we observe in Figure 6 that Disco has a significantly lower network footprint compared to a centralized approach. For decomposable distributed aggregations, the network consumption of the central approach scales linearly with the height of the network tree (see Figure 6a). In contrast, the network consumption of Disco stays nearly constant. Disco is independent of the height of the network topology, as all raw events are sent exactly once from a sensor to a child node. Beyond that, the intermediate nodes only exchange one partial aggregate and some window metadata per second. This causes an additional network traffic of only 1 MB per node level. As a consequence, a network height of five levels causes up to 6x less network traffic (2GB) compared to the centralized approach (12GB).

For holistic aggregations (see Figure 6b), Disco needs to send all raw events to the root node. Already for a tree of height two, we observe that Disco sends fifty percent more data than for decomposable aggregations. However, by sending events as groups of slices instead of individually, we avoid the additional TCP overhead of a purely centralized approach. While all single events are sent at each level for a centralized approach, slices become the smallest message unit in the distributed approach. In this scenario, we can save 50% per level compared to the centralized approach, which has a large effect once the tree becomes significantly deep. For five levels, we send only 7.5 GB of distributed slice data compared to 12.4 GB centralized single events.



(a) avg aggregate function (b) `median` aggregate function

Figure 6: Network cost for different aggregations in Disco.

Summary. Sending raw events is the dominant factor of the network cost. In Disco, we avoid sending individual events between nodes, which results in a high reduction of network traffic. For decomposable functions, Disco completely avoids sending individual events, which drastically reduces the network load. Even for holistic functions, we reduce TCP overhead by combining and sending events in slices.

5 CONCLUSION

In this paper, we present Disco, a distributed complex window aggregation approach. Disco combines the distributed data aggregation concepts from wireless sensor networks with the complex windowing and aggregation semantics from modern stream processing engines. This allows us to reduce network traffic while providing efficient aggregations on arbitrary windows. Our evaluation shows that Disco's throughput scales linearly and that Disco significantly reduces network costs compared to a centralized approach. With the ever increasing number of sensors in the IoT, Disco lays a foundation for efficient, application transparent, distributed stream processing.

Acknowledgements: Work partially funded by DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5) and the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A and 01IS18037A)

REFERENCES

[1] Tyler Akidau et al. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB* 8, 12 (2015), 1792–1803.
[2] Paris Carbone et al. 2015. Apache Flink(TM): Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.
[3] Paris Carbone et al. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*. 1201–1210.
[4] Jim Gray et al. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *DMKD* 1, 1 (1997), 29–53.
[5] Paulo Jesus et al. 2015. A Survey of Distributed Data Aggregation Algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2015), 381–404.
[6] Jeyhun Karimov et al. 2018. Benchmarking Distributed Stream Processing Engines. In *ICDE*. 1507–1518.
[7] Sailesh Krishnamurthy et al. 2006. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*. 623–634.
[8] Jin Li et al. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record* 34, 1 (2005), 39–44.
[9] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. 311–322.
[10] Samuel Madden et al. 2002. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*. 131–146.
[11] Kanat Tangwongsan et al. 2015. General incremental sliding-window aggregation. *PVLDB* 8, 7 (2015), 702–713.
[12] Ankit Toshniwal et al. 2014. Storm@twitter. In *SIGMOD*. 147–156.
[13] Jonas Traub et al. 2018. Scotty: Efficient window aggregation for out-of-order stream processing. In *ICDE*. 1304–1307.
[14] Jonas Traub et al. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*. 97–108.
[15] Yong Yao et al. 2002. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record* 31, 3 (2002), 9–18.
[16] Matei Zaharia et al. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*. 423–438.
[17] Steffen Zeuch et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*.