Desis: Efficient Window Aggregation in Decentralized Networks

Wang Yue
Hasso Plattner Institute
University of Potsdam
Germany
wang.yue@hpi.de

Lawrence Benson
Hasso Plattner Institute
University of Potsdam
Germany
lawrence.benson@hpi.de

Tilmann Rabl
Hasso Plattner Institute
University of Potsdam
Germany
tilmann.rabl@hpi.de

ABSTRACT

Stream processing is widely applied in industry as well as in research to process unbounded data streams. In many use cases, specific data streams are processed by multiple continuous queries. Current systems group events of an unbounded data stream into bounded windows to produce results of individual queries in a timely fashion. For multiple concurrent queries, multiple concurrent and usually overlapping windows are generated. To reduce redundant computations and share partial results, state-of-the-art solutions divide windows into slices and then share the results of those slices. However, this is only applicable for queries with the same aggregation function and window measure, as in the case of overlaps for sliding windows. For multiple queries on the same stream with different aggregation functions and window measures, partial results cannot be shared. Furthermore, data streams are produced from devices that are distributed in large decentralized networks. Current systems cannot process queries on decentralized data streams efficiently. All queries in a decentralized network are either computed centrally or processed individually without exploiting partial results across queries.

We present Desis, a stream processing system that can efficiently process multiple stream aggregation queries. We propose an aggregation engine that can share partial results between multiple queries with different window types, measures, and aggregation functions. In decentralized networks, Desis moves computation to data sources and shares overlapping computation as early as possible between queries. Desis outperforms existing solutions by orders of magnitude in throughput when processing multiple queries and can scale to millions of queries. In a decentralized setup, Desis can save up to 99% of network traffic and scale performance linearly.

1 INTRODUCTION

We are witnessing an explosion of data generated in many domains, from e-commerce [5, 7] to social media [20, 48] and industrial process control [29, 50, 61, 63]. The massive amounts of data are produced in form of large-scale and high-velocity continuous data streams. Current SPEs such as Apache Flink [14], Apache Spark Streaming [64], and Apache Storm [57] are developed to perform efficient data aggregation and can process millions of events per second [26, 47], which is why they gained wide adoption in industry and research [28]. These SPEs split unbounded data streams into bounded windows and then process each window separately [4]. Processing of interesting streams often involves many simultaneous queries with the same keys and selection predicates. This produces many concurrent windows. Depending on the characteristics of queries, concurrent windows are created with different window types, aggregation functions, and window measures. This results in significant overlap of windows within but also across queries. Processing concurrent windows individually leads to redundant computations and creates unnecessary resource consumption and performance degradation [12, 52].

To avoid redundant computations, current research systems utilize window slicing to cut overlapping windows into slices and compute partial aggregations [9, 23, 24, 32, 33, 38, 39, 56]. They combine the partial results of slices instead of calculating overlapping windows individually. However, these solutions are limited to sliding windows and tumbling windows, in which the window sizes are fixed. But for unfixed-sized windows, e.g., session windows and user-defined windows, they cannot split windows and share overlaps. To overcome this limitation, Traub et al. [58–60] presented Scotty, which can apply window slicing to all window types. Scotty is limited to sharing partial results between windows that have the same aggregation functions. For example, if there are two windows that calculate sum and average functions, Scotty processes two windows individually even if the windows overlap and both use sum aggregation functions.

Current data streams are often distributed on massive amount of devices [35, 36, 49]. They constitute large-scale and high-velocity decentralized networks. Current designs for SPEs can efficiently work with a limited number of data streams in a centralized network. To process data from decentralized networks, current SPEs [14, 57, 64] are deployed in data centers and collect distributed data from decentralized data sources. Therefore, all data produced in decentralized data streams are required to be collected [1]. This can lead to very high network costs for large-scale setups.

To reduce network overhead and improve performance, several research proposals [10, 43, 44, 66] off-load part of the data processing that would be performed in data centers to devices closer to data streams. Instead of sending every event, in these deployments devices produce and transfer partial results. Data centers then aggregate partial results to output final results. However, current solutions cannot efficiently process workloads with multiple queries in a decentralized deployment. Disco [10] is designed to process windows in a decentralized fashion and employs Scotty to process multiple queries on edge devices. It pushes down all queries to edge devices and creates and processes concurrent windows there. Disco has intermediate nodes that aggregate partial results from edge devices and send aggregated partial results to its data center. For overlapping windows, Disco employs Scotty to avoid duplicate calculations and to send partial results of windows. As it is based on Scotty, Disco can also only share partial results between windows with the same aggregation functions. Also, window slicing is applied only on edge devices

and partial results from overlapping windows are transmitted to and processed by intermediate nodes individually. This redundant computation and transmission decreases performance and increases the network cost of SPEs.

In this paper, we present Desis, an SPE designed to share computation between concurrent windows with different window measures, window types, and aggregation functions. Desis can efficiently process parallel queries with decentralized aggregation on edge and fog computing environments. The main contributions of our paper are:

(1) We design and implement Desis, a stream processing system that efficiently does multi-query processing in decentralized and centralized networks.
(2) Desis features an aggregation engine that can process windowed queries with the same keys and selection predicates and share partial results between windows across different window types, window aggregation functions, and window measures.
(3) Desis' aggregation engine applies decentralized aggregation to push down window aggregation to all the nodes and processes concurrent windows in decentralized deployments.
(4) Our extensive experiments in both decentralized and centralized scenarios show that Desis can improve performance by orders of magnitude with respect to throughput and network overhead.

The rest of the paper is structured as follows. Section 2 introduces the foundation of stream processing that Desis is built upon. In Section 3, we present an overview of Desis. In Section 4 and Section 5, we present the technical details of our aggregation engine and decentralized aggregation. We finally evaluate Desis in Section 6 and discuss related work in Section 7.

2 BACKGROUND

Stream processing engines process queries on long continuous data streams and output results periodically. Depending on different use-cases, SPEs get a single or multiple data streams as input and have to process multiple queries simultaneously. In this section, we present some background in stream processing and discuss the features of concurrent windows and decentralized networks.

2.1 Windowing

To process queries within data streams, SPEs utilize windows to divide infinite continuous data streams into finite data collections. The window type is for SPEs to describe the windows from continuous streams. In the Dataflow model [3], Akidau et al. define three window types: tumbling windows, sliding windows, and session windows. A tumbling window is defined by a length l and divides the data stream into equally sized windows of this length. The beginning of the new window is the end of the previous window. A sliding window is specified by a step size s and a length l. s determines the direct distance between the beginning of the new window and the beginning of the previous window, l the length of the window. When s is equal to l, the sliding window is a tumbling window. In the case when s is smaller than l, there are consecutive overlaps between windows. In a session window, the data stream is divided into an activity period and an inactivity period. When no events arrive for a time l_g (gap) after the last event, the window is ended and enters the inactivity period. Examples of session windows are browser sessions and ATM interactions. Besides the three window types described above, there are also user-defined windows. User-defined windows are determined by special events that mark the window start and end.

The window type specifies when to start and end windows, and the window measure decides how to measure events in a window. A window can be determined by different measures, e.g., based on time or count [13]. The definition of a count-based window is similar to that of a time-based window, i.e., where the length l in a count-based tumbling window is the number of events per window. In addition, events from data streams usually have different keys, e.g., speed, temperature, and humidity and different queries have different selection predicates, e.g., `WHERE speed = 80 KM/h` and `WHERE Temperature > 25`. Events with different keys and different selections have to be added to individual windows, and those windows need to be processed separately. We use window keys to mark windows that involve different keys and selection predicates.

2.2 Aggregation Function

Aggregation functions define how to calculate events in a given window. Gray et al. classify aggregation functions into three types distributive, algebraic, and holistic based on their calculation method [21]. Distributive aggregate functions are able to first calculate partial results for sub-parts of data and later merge them to final results. A function *F()* is distributive, if there exist a function *G()*, so that $F(X_{0..n}) = G(F(X_{0..i}), F(X_{i..n}))$ is true, e.g., *sum*, *count*, and *min*. For algebraic aggregate functions, there are functions *G()* and *H()*. The function *G()* returns an m and passes it to *H()* as an argument. We let *F()* be distributive, and an algebraic function is $F(X_{0..n}) = H(G(F(X_{0..i}), F(X_{i..n})))$. Algebraic aggregate functions can be computed from arguments, which are from distributive aggregate functions, e.g., *avg* (as *sum* / *count*). Holistic aggregate functions cannot be represented by partial results and require all events to determine the result, e.g., *median* or *quantiles*. Jesus et al. present terminologies that describe aggregation function as decomposable, self-decomposable, and non-decomposable [30]. Self-decomposable functions can be directly split and work on subsets of data. A decomposable function consists of self-decomposable functions, it needs more steps to calculate results than a self-decomposable function. Self-decomposable functions refer to distributive functions and decomposable functions refer to algebraic functions. For ease of understanding, we merge the concept of self-decomposable function into decomposable. Decomposable functions include count, sum, max, min, and average. Non-decomposable functions are the same as holistic functions and cover median and quantile.

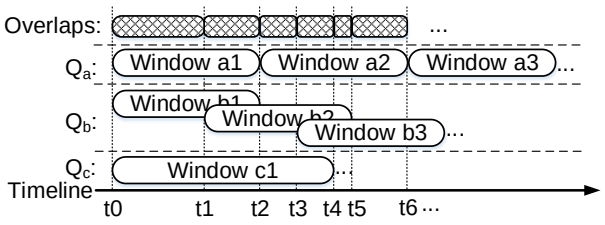


Figure 1: Overlap between Q_a, Q_b, and Q_c.

2.3 Window Overlap

When processing multiple queries, systems have to create and maintain concurrent windows. Once a window ends, all events in

that window are aggregated. In Figure 1, there are three queries, Q_a, Q_b and Q_c, their window types are tumbling, sliding, and session, respectively. We first create windows according to window types separately at t_0. We observe that between t_0 and t_1, there is an overlap between window a_1, b_1 and c_1. From t_2 to t_3, there is an overlap between window a_1, window b_1, window b_2, and window c_1. All windows ingest events from the same data steams and process the same events in overlaps. If we aggregate those windows individually, overlaps in between will be repeatedly calculated, which is a waste of CPU resources. Besides, an event is dropped only if this event has already been calculated by all windows it belongs to. The duration of windows differs, which leads to different time spans that are buffered in memory. For large windows, events are kept in memory for a long period and consume a lot of memory until they are dropped. Therefore, window-slicing techniques are utilized to share window overlaps.

2.4 Window Aggregation in Decentralized Networks

In decentralized setups, data streams are distributed across different nodes. Current SPEs can execute either centralized or decentralized aggregation to aggregate data. In centralized aggregation, the topology of the network structure includes a center and many data streams. We denote the center as the root node, which serves as the central node of a decentralized network. All data are collected from data streams and transferred to the root node. The root node performs all window aggregations.

In decentralized aggregation, the topology is divided into three types of nodes, which enables systems to distribute the computation across nodes instead of loading all data into one root node [10]. As shown in Figure 2, there are a root node, intermediate nodes, and local nodes. In general, there are multiple local nodes and intermediate nodes but only one root node in a topology. Local nodes can connect to the root node directly or via intermediate nodes. For complex networks, more intermediate nodes are interconnected between the local and root nodes, and there are more hops from a local node to the root node.

To process queries, local nodes create individual windows for every query to collect data from data streams and then perform aggregations to output partial results. Afterwards, intermediate nodes create windows for each query again to aggregate partial results from their child (local or intermediate) nodes and send new partial results to their parent node (intermediate or root). Finally, the root node builds windows and performs final aggregations to output the final results.

3 SYSTEM OVERVIEW

In order to perform aggregations across a large number of queries, we propose Desis. Desis is able to process multiple queries from data streams that are distributed in decentralized networks.

3.1 Components of Desis

As shown in Figure 2, Desis has five components: interface, query analyzer, window manager, aggregation engine, and message manager. As the key component, we discuss the aggregation engine in detail in Section 4. The user interface provides APIs for users to invoke commands and pass queries into Desis. All queries passed from the interface are collected in the query analyzer (QA). The QA analyzes queries and emits window attributes of each query so that the system is able to create windows based

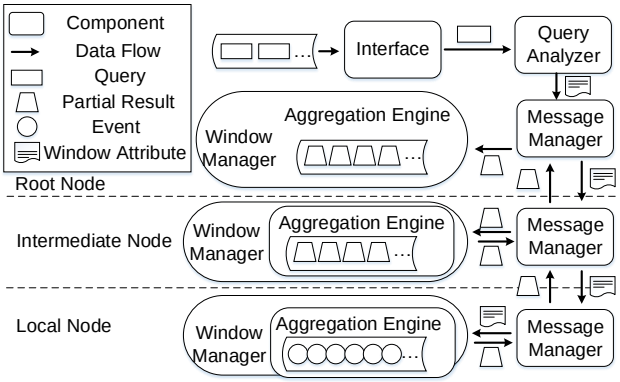


Figure 2: System architecture.

on them. The window attributes include keys, window types, window measures, and aggregation functions. The window manager (WM) processes the actual windows. It starts and ends windows based on their window type and measure, and then executes windows via the aggregation engine. The message manager is used to communicate between nodes, as Desis is distributed in decentralized networks.

3.2 Fault Tolerance

Desis supports basic fault tolerance and can add and remove queries and nodes during runtime. To add or remove a node, users have to inform the root node of the IP address and port of the node. The root node then adds or removes the node from the Desis cluster and sends the new topology to all other nodes. Desis has a timeout for every local and intermediate node. When a node loses connection, Desis will remove this node from the cluster and inform users. Desis can receive new queries and pass queries to the query analyzer and every query has a unique ID. The aggregation engine will receive the window attributes from the query analyzer and process new queries. To remove a running query, users have to provide the query ID and waiting time to remove queries immediately or wait for the last window to end.

4 AGGREGATION ENGINE

In this section, we present the *aggregation engine*, which enables sharing computation across multiple queries, windows, and aggregation functions. To process multiple queries, current SPEs, such as Flink [14], Spark [64], and Storm [57], create multiple concurrent windows, one for each query. These windows are executed individually, even though the events in the windows are the same. This causes redundant computation, even if windows share the same window properties and aggregation functions. Advanced approaches, like Scotty [60] and Cutty [15] are able to slice windows and aggregate events that are in slices to output results. They can share partial results between windows with different window measures and types, including tumbling, sliding, session, and user-defined windows. But for windows that have different aggregation functions, they cannot share results.

To allow for sharing across aggregation functions, our aggregation engine ensures that each event is processed only once for multiple windows that have the same keys and selection predicates. Instead of creating and executing windows for each query, the aggregation engine first cuts windows into slices and converts aggregation functions into operators. The aggregation engine shares partial results of slices between different windows

with different window types, window measures, and aggregation functions. As shown in Figure 2, the aggregation engine is used in Desis to assist the window manager to process windows. In decentralized setups, an aggregation engine is deployed on each node to support pushing down windows for local processing.

4.1 Window Slicing

To avoid redundant computation between concurrent windows, we use window slicing. It reuses partial results of each overlap instead of processing all windows individually. This technique not only improves the performance of our system but also reduces the consumption of computational resources. Before describing our approach, we define the terms query-group and punctuation.

Query-Group: a query-group is a set of queries that partial results can be shared between and in which every event is processed only once. For multiple queries, current systems create separate windows to process events. Our aggregation engine puts queries that can be shared into one query-group and produces slices instead of windows.

Punctuation: we use the term punctuation to identify when to create or terminate a window. Each window has two punctuations, a start punctuation (*sp*) and an end punctuation (*ep*). The start punctuation marks the creation of a window and the end punctuation marks a window end.

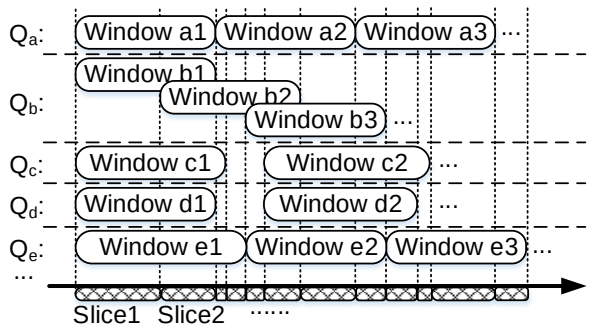


Figure 3: Windows slicing with multiple window types.

In Figure 3, we show how the aggregation engine processes multiple queries. Q_a has tumbling windows and there are no overlaps. Q_b has sliding windows and new windows are created before old windows end, causing overlaps between windows. Q_c and Q_d contain non-fixed sized session and user-defined windows. Q_e contains a tumbling count-based window and its windows can be sliced as tumbling time-based windows. All five queries share the same aggregation function while having different window types, and queries are all in one query-group. The aggregation engine slices concurrent windows into several slices based on their *sp* and *ep*. Regardless of how many concurrent windows are processed, the aggregation engine creates a slice if there is an *sp*. Slices are terminated when the aggregation engine meets an *ep* or *sp*. The aggregation engine then outputs the partial results of that slices. For tumbling, sliding, and counting windows, punctuations are affected by time and count measures and they have fixed window sizes. For user-defined and session windows, window ends are dynamic as they are terminated with user-defined events or gaps where there are no events.

4.2 Window Merging

To output results of every window, the aggregation engine performs incremental aggregation for every event that arrives in a slice. Once a window ends, the aggregation engine assembles all slices based on their original windows. Given a slice, current solutions are able to reuse computation results only if all windows involved in this slice have the same aggregation functions. However, for multiple queries with different functions, current window-slicing solutions still have to compute the partial results of each slice individually [60]. To deal with that, our aggregation engine breaks down aggregation functions into operators and shares those operators between different slices.

Aggregation Function	Operator
sum	*sum*
count	*count*
average	*sum*, *count*
product	*multiplication*
geometric mean	*multiplication*, *count*
max	*decomposable sort*
min	*decomposable sort*
median	*non-decomposable sort*
quantile	*non-decomposable sort*

Table 1: Relationship between aggregation functions and operators

4.2.1 Aggregate Operators. Operators are the most basic aggregation functions that a stream aggregation is broken down into (Section 2). Instead of processing aggregation functions, the aggregation engine shares operators between slices and executes operators to provide intermediate results. When a slice is terminated, the intermediate results from different operators are combined to output partial results of this slice.

We support many operators, e.g., *count*, *sum*, *multiplication*, *square root*, *decomposable sort*, and *non-decomposable sort*. Given two overlapping windows that calculate an average and a sum, the average function can be broken into the sum operator and count operator. All the windows have to execute the sum operator and the aggregation engine can share this operator. For each event, the aggregation engine performs only two aggregations (count and sum) instead of three (count, sum, and sum). For overlapping windows that have multiplication and geometric average functions, we can share multiplication operators between overlapping windows. We define decomposable sort and non-decomposable sort operators. Decomposable sort executes sort incrementally and drops computed events. It can be shared between max and min. Non-decomposable sort keeps all events and performs a final sort when the slice is ended. Its results can be shared between max, min, median, and quantile.

In Table 1, we show the relationship between aggregation functions and operators. Instead of executing separate aggregation functions for windows, we utilize operators to share intermediate results between different functions. Also, for complex aggregation functions, users can define new operators to break down functions.

4.2.2 Multiple Aggregation Functions. Once a slice is terminated, the aggregation engine has to calculate results by executing aggregation functions. Regardless of window types, there are two scenarios: single aggregation function and multiple aggregation functions. The former scenario is discussed in the previous solutions [60] and Cutty [15] and here we show the latter scenario in Figure 3. The first two slices (*slice1* and *slice2*) are shared among five queries. We let the aggregation functions of Q_a and Q_b be max and median. The aggregation functions of Q_c, Q_d,

and Q_e are sum, count, and average. Our aggregation engine shares the *non-decomposable sort* operators between Q_a and Q_b. Also, *sum* and *count* operators are shared between Q_c, Q_d, and Q_e. With operators, we avoid individual computation of each aggregation function and concurrent windows that have different aggregation functions can share partial results.

4.2.3 Non-Aggregate Operators. To filter out events required by different queries, Desis provides additional operators. There are two non-aggregate operators: selection and deduplication. The selection operator can select events matching given predicates. The deduplication operator drops events that are duplicated. For queries that have non-overlapping selection predicates, the aggregation engine puts them into the same query-group, e.g., `WHERE speed > 80 KM/h` and `WHERE Speed < 25 KM/h`. The aggregation engine can create multiple non-aggregate operators for every window. When a slice is created, the aggregation engine first checks all windows the slice is from and binds operators of those windows to the slice. All selection operators are processed separately and they all have individual results. In this case, every event is still processed only once and the aggregation engine does not need to maintain the context of many groups, i.e., the list of slices and aggregation operators. If there are queries that have the same selection predicates and keys, they will be put into the same query-group as their results can be shared between each. When processing queries that have overlapping selection predicates or keys, queries are put into different query-groups.

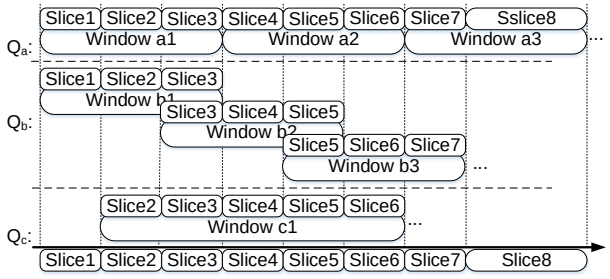


Figure 4: Aggregation engine processing multiple queries.

4.3 End to End Optimization

We now discuss the high-level query processing of our aggregation engine. In Figure 4, we pass three queries into our system. Q_a has tumbling windows with a max aggregation, Q_b has sliding windows and a quantile function. Query Q_c contains session windows and a median function. The aggregation engine first puts the three queries into a query-group and converts their aggregation functions into operators. For queries that have different keys or selection predicates, they can also be put into the same query-group if their selections fully overlap or do not overlap. For each incoming event, our aggregation engine performs incremental aggregations and different selection operators have their own partial results. Whenever there is a punctuation, the aggregation engine terminates the current slice and executes operators instead of aggregation functions. For example, *slice2* is shared between window *a1* (Q_a), window *b1* (Q_b), and window *c1* (Q_c) and has to calculate the max, quantile, and median. To calculate the partial results of *slice2*, the aggregation engine executes a *non-decomposable sort* operator. Afterwards, the aggregation engine puts the partial results of this slice into a list and creates the next slice. Each slice is marked to which original windows it belongs, e.g., *slice2* is mapped to window *a1*, window *b1*, and window *c1*.

Once there is an *ep*, the aggregation engine has to terminate a window. The aggregation engine first iterates over the list and aggregates all partial results that belong to the window corresponding to that *ep*. For example, we aggregate the partial results of *sclice1*, *sclice2*, and *sclice3* to output the results of window *a1* and window *b1*. Also the result of window *c1* is assembled from *slice2*, *slice3*, *slice4*, *slice5*, and *slice6*. The results from different selection operators are processed and produced individually. In addition, if there are any partial results that do not belong to any window, the aggregation engine will delete them from the list.

5 DECENTRALIZED AGGREGATION

In this section, we present Desis in a decentralized setup and show how it executes concurrent queries with decentralized aggregation. In decentralized networks, data streams arrive at different nodes. Current stream slicing approaches, like Scotty [60] and Cutty [15] must collect this data to a single node to aggregate it. To process data decentrally, Disco [10] moves window aggregation to the local nodes and utilizes Scotty to share partial results on each node. However, Disco cannot efficiently share results between unfixed-size and fixed-size windows. When processing a large unfixed-size window and many other windows, the local node needs to keep a huge number of slices for a long time. Also, the local node has to traverse these slices to calculate results once the window ends. In addition, Disco only applies the window-slicing technique in local nodes. Disco has to send partial results per window to intermediate or center nodes even though those results are from overlapping windows. Also, the overlapping windows are processed individually on intermediate and center nodes without sharing results. Compared to current solutions [10, 15, 60, 62], Desis performs window slicing on all nodes. Desis pushes down slices instead of windows to local nodes and assembles the final results incrementally on the root node. It is able to share partial results between different concurrent windows on all nodes. Also, Desis does not send partial results per window but per slice, which further reduces the amount of data that is sent through the network. Furthermore, Desis can share partial results between windows with different aggregation functions. In the following, we present how decentralized aggregation processes different window types for decomposable and non-decomposable aggregation functions.

5.1 Decomposable Decentralized Aggregation

To reduce the computational load on the root node and the overall network traffic, Desis performs decentralized aggregations that moves calculations and window slicing to all nodes. In local and intermediate nodes, instead of processing concurrent windows individually, Desis creates slices and aggregate them when they end. It then sends the partial results to the parent node instead of raw events. To distinguish between slices on the root, local, or intermediate nodes, we call them root slices, local slices, and intermediate slices, respectively.

5.1.1 Fixed-Sized Windows. Decentralized aggregation works well with concurrent windows that have decomposable functions. If we process only one tumbling or sliding window, windows built on different local nodes end at the same time. This is because windows are time-based and their sizes are fixed. Concurrent sliding and tumbling windows are also sliced into fixed-sized

slices. All local slices are created and terminated at the same time even though they are from different local nodes.

On intermediate and root nodes, the system has to collect partial results from child nodes and merge them. However, systems may face missing or duplicated slices. To overcome this, we use an auto-incrementing id for each local slice. When a new local slice is created, its id is the increment of the previous slice id. We mark every partial result generated from local slices with its slice id, so intermediate and root nodes can merge partial results by aggregating results that have the same slice id. The intermediate and root nodes reuse the slice id of the partial results they calculated as their own ids and their results are marked with ids as well. The length of an intermediate slice is equal to the number of child nodes connected to it. Similarly, the length of a root slice is the same as the number of children the root node has.

5.1.2 Unfixed-Sized Windows. When processing session windows, a window ends when there is a long gap. The data in each data stream is different, which leads to different gaps at different points in time. In this case, one local node can produce more session slices than others, and the system cannot recognize which session slices belong to the same session window. For user-defined windows, the window ends depend on user-defined events and the exact time of that events is unpredictable. One such use case for a user-defined window would be computing the maximum speed of cars for each trip. In local nodes, slices are created to collect speed values for the car and the maximum speed is calculated once the trip is done. In this case, slice sizes of local slices are not the same, as slices for trips with different lengths are not finished at the same time. Also, the final results need to cover different numbers of partial results from different nodes. So the slice sizes of both session windows and user-defined windows are not fixed and different local nodes have different slice sizes. To deal with session windows, our system lets each session slice carry the start time and end time of its session gap. The root node records the latest session gaps for each child node. When all session gaps from different child nodes cover each other, the session window ends and all session slices are merged to output the final results. For user-defined windows, we introduce a watermark that allows the root node to terminate user-defined windows timely and not be delayed by partial results from long slices. The watermark also can be used by session windows to terminate long session windows. Based on the above discussion, our system can put fixed and unfixed windows into one query-group and slice them at local nodes. The lengths of intermediate and root slices are the number of their child nodes. Also, when session gaps from child nodes cover each other or watermarks are encountered, intermediate and root slices are terminated as well. If a window ends, the aggregation engine will merge all slices from that window to output the final results.

5.1.3 Decentarlized Aggregation with Fixed and Unfixed-Sized Windows. In Figure 5, we show three example queries and they are all time-based, Q_a has tumbling windows and executes sum functions. Q_b and Q_a execute average functions and they have sliding and user-defined windows.

On the root node, the system puts Q_a, Q_b, and Q_b into a query-group and outputs window attributes. All windows produced from that query-group can be shared, as they consist of the two operators *sum* and *count*. Then, the message manager distributes the window attributes of this query-group to local and intermediate nodes.

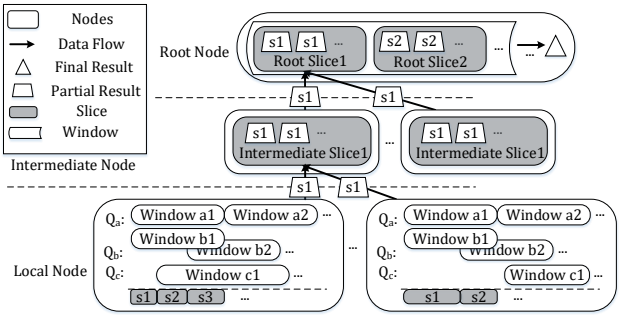


Figure 5: Decentralized aggregation processing a decomposable function.

On a local node, the aggregation engine splits windows into local slices according to the window attributes that are received from the root node. Then, the aggregation engine executes incremental aggregations for every event. Otherwise, the aggregation engine may get stuck since it has to spend a long time iterating and computing all events in the slice. Once a local slice ends, the aggregation engine executes its aggregation functions and transmits partial results to the intermediate nodes. Also, if a local slice is ended by an end punctuation (*ep*), the system will mark this slice with the *ep*.

On an intermediate node, our system creates an intermediate slice and collects partial results from its child nodes for this slice. When the intermediate slice receives all results from local nodes, it calculates partial results and sends them to the root node. If partial results from local nodes are marked with a *ep*, the intermediate partial result will keep this *ep*. Similar to local nodes, we also perform incremental aggregations on intermediate nodes. For example, in Figure 5, when there are two partial results of *s1*, they are collected in the *intermediate slice1*. The intermediate node ends the *intermediate slice1* and executes the underlying operators.

On the root node, partial results from its child nodes are gathered and incrementally calculated into a root slice. When a slice ends, the aggregation engine executes aggregation functions and saves its result. If intermediate partial results contain an *ep*, the aggregation engine has to assemble all partial results belonging to this window and output the final result. In Figure 5, we terminate the root slice when both partial results of intermediate slices arrived. The partial result of *s2* contains an *ep*, and the root node ends the window according to this *ep*.

5.2 Non-Decomposable Decentralized Aggregation

For non-decomposable functions, e.g., median and quantile, all events have to be collected from data streams to the root node. This is needed because local nodes cannot perform partial aggregations and output partial results. In a centralized aggregation, all events are sent to the root node individually, which incurs a high network overhead. To reduce network overhead, current systems first batch events before transferring them [17]. They set a unified batch size and create windows on local nodes to batch events. When the window size is equal to batch size, they end the window and send it to parent nodes. The window sizes are fixed and windows act as count-based tumbling windows. In root nodes, the system has to first check each event to find the window end and then sort events to give the final results.

Although they can reduce network overhead, they centralize all computation in the root node.

To process non-decomposable functions efficiently, our system creates slices on local and intermediate nodes. Once a slice ends, the local node batches all events in this slice and sends them to its parent node. The root node collects all slice batches and put them in its window and performs aggregation. So, for each arriving batch, the root node behaves the same as process decomposable functions but without incremental aggregations. We mark the slice batch if there is a *ep*. The root node terminates windows only if the arrived slice batch carries *ep*. As sorting is necessary for both median and quantile, the root node has to execute sorting anyway. In our system, the local and intermediate nodes execute non-decomposable sort operator for non-decomposable functions so that the root node process sorted events.

Count-based windows cannot be processed locally, since only the root node terminates count-based windows. For decentralized aggregation, our system puts windows with decomposable functions and count-based windows into different query-groups and processes them separately. However, count-based windows can be put into the same query-group as windows with non-decomposable functions because they are all calculated on the root node. If we use centralized aggregation to process all windows, count-based windows and all other windows can be put into the same query-group and share partial results.

6 EVALUATION

In this section, we present our experimental design, evaluate the performance of Desis, and compare it to state of the art.

6.1 Experimental Design

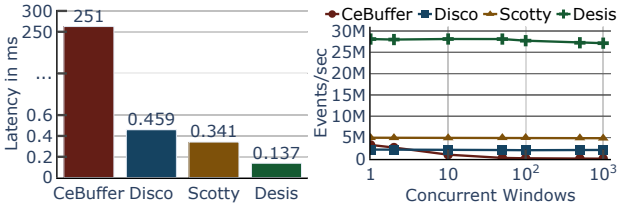
We conduct experiments on a 10-node cluster with 25G Ethernet. Each node has two 18-Core Intel Xeon Gold 5220S CPUs and 96 GB main memory. All nodes are running Ubuntu 20.04 and OpenJDK 1.8.0.312 64-bit. In our experiments, we measure throughput, network overhead, and latency for Desis and relevant baseline approaches. We report sustainable throughput [31], i.e., the throughput a system can process without an ever-increasing backlog. We only consider sustainable network overhead in our experiments. We calculate network overhead for all nodes in the cluster including intermediate nodes. In decentralized networks, there are commonly multiple intermediate nodes ('hops') between edge devices and the data center. Even though intermediate nodes only transfer data and do not process it, we cannot remove these nodes. We evaluate the event-time latency instead of process-time latency, which calculates the time from when the event is created to when the result involving the event is produced. In this case, we can avoid coordinated omission [55] that leads to a significant underestimation of latency [18].

6.1.1 Baselines. Besides Desis, we evaluate five baselines, central buffer (CeBuffer), Scotty [60], Disco [10], Desis bucket (DeBucket) [40, 41], and Desis Sharing Windows (DeSW). All baselines are implemented in Java. CeBuffer creates buffers for each window and collects events to window buffers. When windows end, it performs window aggregation. CeBuffer does not perform incremental aggregation and every window has its own buffer. The Scotty baseline is developed based on the Scotty API, it can perform incremental aggregation and share partial results between windows that have the same aggregation functions. Disco is a decentralized system that uses the Scotty API. We modify the data input of Disco to use our data generator. Disco can perform incremental aggregations and perform window slicing on local nodes. To make a fair comparison and measure the improvement of Desis, we develop DeBucket and DeSW. DeBucket creates buckets for every concurrent window and calculates arriving events incrementally, but all the buckets are processed separately even though there are overlapping buckets. DeSW is similar to Scotty, which can process windows that have the same aggregation functions and window measures. Both DeBucket and DeSW are developed based on Desis and they have the same architecture that can calculate decentralized aggregations. Compared to Debucket, DeSW creates buckets for different functions. CeBuffer and Scotty are centralized systems, and only one node processes all events. They can be deployed on the same topology as Desis, but only the root node processes events. Other nodes in networks collect events from data streams or their child nodes and then send data to parent nodes directly.

6.1.2 Generators. To simulate a decentralized network environment, we generate data by replaying recorded data from a synthetic dataset and we let the data generators read from different positions in the data set to simulate different data streams. Events in the data stream have four fields, *time*, *key*, *value*, and *event*. The data generator gives every event a timestamp. The event values are from the DEBS 2013 dataset [46]. The field *event* represents user-defined events of user-defined windows. The data generator is configured with the key distribution, selection predicates, the frequency of user-defined events and window gaps. Our query generator can provide arbitrary queries with different keys, window types, aggregation functions, window measures, and window sizes. The query generator is configured with the distribution of the window keys, window length, window types, window measures, and aggregation functions. In a decentralized setup, we deploy a data generator on each local node and the query generator on the root node.

6.2 End to End Performance

We first compare Desis, Scotty, Disco, and CeBuffer to study overall performance. We focus on throughput and latency to measure each system.



(a) Latency of a single window. **(b) Throughput of concurrent windows.**

Figure 6: End to end throughput and latency.

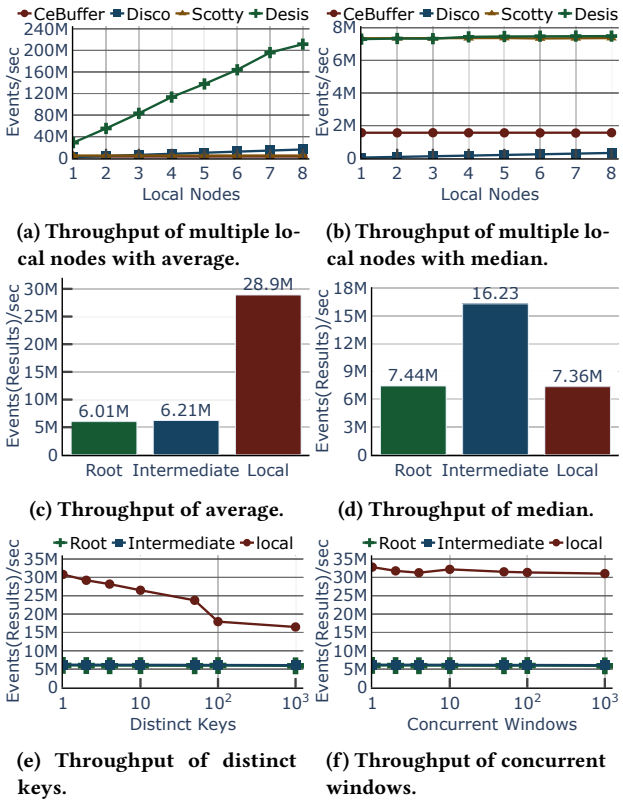
6.2.1 End to End Throughput and Latency. We first measure latency of a single tumbling window query with an average aggregation function with 10 distinct keys. We then execute a set of windows that have equally distributed lengths from 1 to 10 seconds. We gradually increase the number of concurrent windows from 1 to 1000 and measure their throughput. We deploy all systems on a single node.

Results. In Figure 6a, we show latency of each system. We see that CeBuffer has the highest latency. When new events arrive, CeBuffer puts them into a buffer for each window and performs

an aggregation function once the window ends. CeBuffer has to iterate overall events of a window, every time when the window is triggered. The other systems perform aggregation functions incrementally for each event. Disco uses the Scotty API to process events, but it uses a single thread to execute everything, e.g., receiving raw events, processing events, and sending results, which affects its latency. Scotty is developed as a centralized system and it does not send partial results through the network.

The results for queries with concurrent windows are shown in Figure 6b. CeBuffer has a higher throughput than Disco for a small number of concurrent windows, but its performance drops sharply when the number of concurrent windows increases. CeBuffer cannot share partial results between different windows, it has to process windows individually. The other systems can share partial results and avoid redundant computations. The throughput of both Disco and Scotty are stable at around 2 million events/s and 5 million event/s, respectively and change only slightly. Their performance is the same as in the single window setup. Desis has the highest throughput at 28 million events/s even when processing 1000 windows simultaneously. Desis is able to calculate window ends in advance instead of checking each arriving event.

Summary. Desis outperforms all baselines and has the lowest latency. When processing concurrent windows with decomposable aggregation functions, its throughput is over 28 million event/s and 5 times higher than Scotty's.



(a) Throughput of multiple local nodes with average.

(b) Throughput of multiple local nodes with median.

(c) Throughput of average.

(d) Throughput of median.

(e) Throughput of distinct keys.

(f) Throughput of concurrent windows.

Figure 7: Throughput under different setups.

6.2.2 Scalability. We first study scalability of each system, all systems initially work on a 3-node cluster: one local node, one intermediate node, and one root node. We gradually add local nodes to the topology and measure their throughput. All local nodes connect to one intermediate node. All systems execute concurrent tumbling windows with average functions and 10 distinct keys. In the second experiment, we profile the throughput of Desis on each node with different constraints. We increase the number of partial results per node to simulate an increasing number of child nodes.

Results. The results of the scalability experiments with average and median aggregation functions are shown in Figure 7a and Figure 7b. Scotty and CeBuffer have constant throughput with an increasing number of local nodes. They perform all calculations on the root node and all other nodes transfer events to the root. Desis has a higher throughput than Disco since it is able to push down and slice windows on local nodes and share partial results on all nodes. Desis and Disco scale linearly with the number of local nodes when processing average functions since both systems perform decentralized aggregation. For the median function, Desis still outperforms the other systems, but its throughput slightly decreases with more local nodes. The performance of the root node determines the overall throughput, because its root node has to collect and process all events from child nodes.

In Figure 7c(average), we see that the throughput of the root node and intermediate node are at 6.01 million events/s and 6.21 million events/s, respectively. They only create windows to collect partial results from child nodes in the intermediate nodes and root node. Figure 7d shows the throughput of the root node for a median aggregation function. All events are sent to root nodes, so the throughput of the root node also limits the overall performance of Desis.

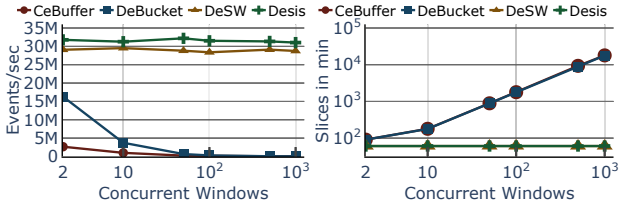
In Figure 7e, we use a single query and vary the number of keys. With more distinct keys the throughput of the local node starts to drop while the performance of the root and intermediate nodes is constant. For multiple queries that have different keys, Desis generate multiple selection operators for every slice. In this case, every event on the local node has to go throughput many selection operators, which decreases throughput. While the other nodes just merge partial results from child nodes. In Figure 7f, we measure concurrent windows with the same keys, and all nodes stay at the same throughput even though they process one thousand windows.

Summary. When processing decomposable functions, decentralized aggregation benefits from adding nodes with respect to throughput. The performance of decentralized aggregation increases linearly with the number of nodes. In addition, one root node or intermediate node can deal with a large amount of child nodes. For non-decomposable functions, the root node is the bottleneck of Desis.

6.3 Optimization Performance

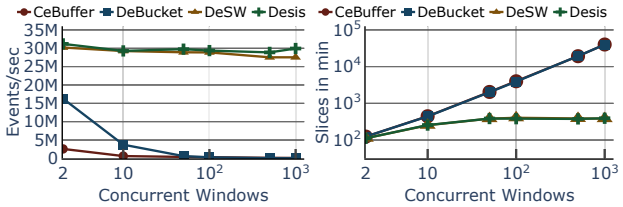
In this experiment, we measure the impact of processing different concurrent windows. Windows have varying window types, aggregation functions, and window measures but use the same keys. We compare throughput between Desis and three baselines, including CeBuffer, DeBucket, and DeSW. DeBucket cannot share any partial results between queries. As a result, DeBucket processes 1000 query-groups individually if we start 1000 queries. DeSW can share partial results between windows that have the same aggregation functions and window measures. The number of individual query-groups for DeSW depends on the number of aggregation functions and window measures. For example,

DeSW will create two query groups if there are only two kinds of functions even though we process thousands of queries. Both DeSW and DeBucket can calculate aggregations incrementally, CeBuffer not.



(a) Throughput of concurrent fixed-sized windows.

(b) Number of fixed-sized slices produced in minute.



(c) Throughput of concurrent unfixed-sized windows.

(d) Number of unfixed-sized slices produced in minute.

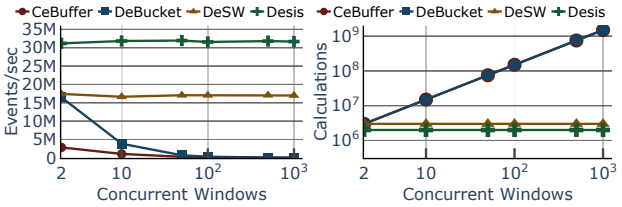
Figure 8: Throughput and number of slices of concurrent windows.

6.3.1 Multiple Queries with Different Window Types. In this workload, we study fixed-sized windows and unfixed-sized windows with varying sizes. We aggregate all windows with an average function but change the window type and window length. We measure throughput and the number of slices.

Results. We first study concurrent tumbling windows, i.e., fixed-sized windows, in Figure 8a. Window lengths are equally distributed from 1 to 10 seconds. For both Desis and DeSW, the throughput does not change with an increasing number windows, but DeSW has a lower throughput than Desis. This is due to the fact that Desis can add all queries to one query-group and process each event only once. For decentralized aggregation, Desis can share results between any windows and it does not check other query-groups. DeSW and other baselines need to move the events to different query-groups that have different aggregation functions or window measures. DeSW has to check if there are other query-groups when new events arrive. In Figure 8b, we measure the number of slices produced from each system per minute. Desis and DeSW cut windows into slices, and every window provides multiple slices. DeBucket and CeBuffer do not perform window slicing, and each window is covered by one slice. DeBucket and CeBuffer have less slices initially, and then the number of slices rapidly increases when more concurrent windows are passed in the systems. For Desis and DeSW, all windows can be fully covered by non-overlapping slices, and they only produce 61 slices every minute even if there are more concurrent windows. Instead of processing events in different concurrent windows repeatedly, Desis and DeSW process every event exactly once, which can avoid duplicated calculations between windows.

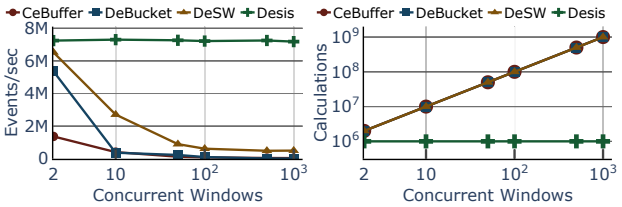
We keep the same setup and change half of the windows to user-defined windows (Figure 8c). We let generators produce 1 user-defined event per second. Compared to the tumbling windows, the window lengths of user-defined windows are small and not fixed, so the aggregation engine has to cut windows in more slices than tumbling windows, and the size of each slice is not predetermined. We see that the performance of all systems slightly decreases due to processing more slices. In Figure 8d, all systems produce more slices when processing unfixed-sized windows, and the slice numbers of Desis and DeSW are constantly at less than 400 per minute.

Summary. Desis and DeSW have higher throughput and fewer slices when processing a large volume of concurrent windows that have the same aggregation functions and window measures. When processing concurrent unfixed-sized, the throughput of systems decreases slightly because they have to create more slices.



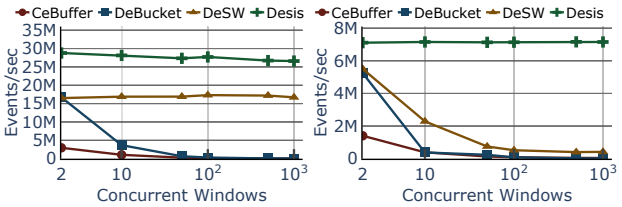
(a) Concurrent windows with average and sum.

(b) Calculations of average and sum in million events.



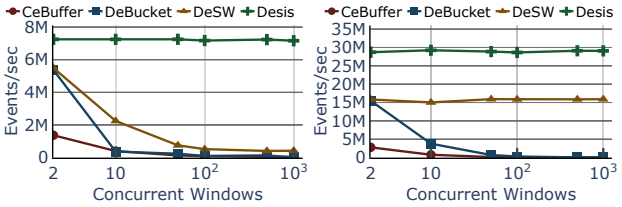
(c) Concurrent windows with different quantiles.

(d) Calculations of quantiles in million events.



(e) Concurrent windows with average and max.

(f) Concurrent windows with average and quantile.



(g) Concurrent windows with quantile and max.

(h) Concurrent time-based and count-based windows.

Figure 9: Throughput and number of calculations of concurrent windows with different aggregation functions and window measures.

6.3.2 Multiple Queries with Different Aggregation Functions and Window Measures. We keep the setup as in the previous workload but change the aggregation functions. All time-based queries perform tumbling windows with 1 second length, count-based windows have a length of 1 million events. We use queries

that have different aggregation functions, including average, sum, max, and quantile. For quantile functions, the quantile values are distributed from 1 to 1000. We first measure the throughput of each system. We then send 10 million events to the systems and measure the number of executed calculations.

Results. Figure 9a shows the throughput of concurrent windows with average and sum functions. DeBucket and CeBuffer have lower throughput, since they have to perform each query individually. DeSW has to create multiple query-groups against different aggregation functions because DeSW cannot share partial results between different functions. Desis breaks down average and sum into two operators, i.e., sum and count (Figure 9b). So, for each incoming event, two operators are executed once, while DeSW performs calculations for each event three times.

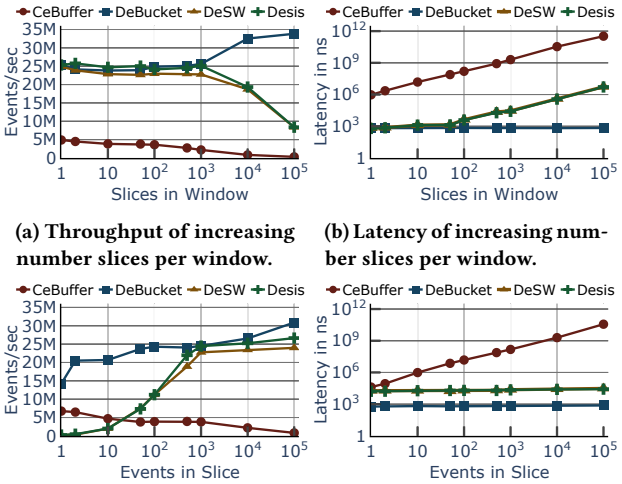
In Figure 9c, systems process concurrent windows with different quantile functions. We aim to simulate situations with multiple different aggregation functions. We see that the throughput of all baselines drops sharply. Desis can constantly process 7 million events/s even with 1000 concurrent windows with different functions. Except for Desis, all systems have to create query-group for each query, and events are repeatedly calculated. In Figure 9d, we observe that Desis adds all queries in one query-group and executes only one non-decomposable sort operator to calculate partial results, which dramatically reduces redundant computations.

We then measure the throughput of combining different aggregation functions and window measures. In Figure 9e and Figure 9f, we calculate two different aggregation functions for each window. Desis needs to perform more operators (i.e., sum, count, and sort), which lightly affects throughput. We measure the combination quantile and max (Figure 9g), its throughput is the same as in Figure 9c as quantile and max can share the same operator and be calculated once, while DeSW has to calculate twice. In Figure 9h, we see that Desis outperforms DeSW when processing windows that have different window measures. DeSW puts count-based and time-based windows into separate query-groups and cannot share partial results between.

Summary. We conclude that Desis can efficiently process multiple queries that have different aggregation functions, and can achieve over 100 times better throughput compared to other systems.

6.3.3 Throughput and Latency of Different Slice Sizes and Window Sizes. In this experiment, we study the throughput and latency of different slice numbers and slice sizes in a window. In Figure 10a and Figure 10b, we vary the number of slices in a window and each slice has 10k events. In Figure 10c and Figure 10d, we vary the number of events in a slice and produce 1k slices in a window. For a fixed number of slices and events, we process count-based windows in this workload. DeBucket and CeBuffer do not perform window slicing. Their window size will increase if we increase the slice size and the slice number in the windows.

Results. DeBucket and CeBuffer aggregate all events of a window without any window slicing. In Figure 10a and Figure 10b, we see the throughput of CeBuffer is decreasing. This is because its window size is growing when more slices are included. DeBucket can perform incremental aggregation and its throughput and latency are not affected by the window size. However, when the number of slices is more than thousand (window size is over million), we see an upward trend in its throughput. In this setup, DeBucket creates and terminates fewer windows in a fixed period



(a) Throughput of increasing number slices per window.

(b) Latency of increasing number slices per window.

(c) Throughput of increasing number events per slice.

(d) Latency of increasing number events per slice.

Figure 10: Throughput and Latency of Systems under Different setups.

of time, which reduces CPU utilization and memory. Desis and DeSW have to aggregate all slices at the window end, which means with higher number of slices the throughput drops and latency increases. This is because Desis and DeSW have to go through and aggregate all slices from this window.

In the next experiment, we fix the number of slices per window and vary the slice size in Figure 10c. The throughput of DeSW and Desis is negatively affected by small size slices since they are busy with creating and maintaining a large number of slices instead of processing events. We present the measured latency in Figure 10d, all systems except CeBuffer have constant latency because they perform a constant amount of calculations once windows end.

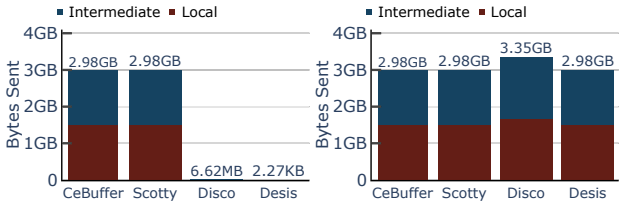
Summary. The window slicing technique cannot be beneficial if we process a large number of slices in a window or if every slice size is very small. Especially when processing concurrent windows that have no overlap, the throughput of the system drops sharply. In this case, Desis will not slice windows and only performs incremental window aggregation.

6.4 Decentralized Performance

We choose CeBuffer, Scotty, and Disco as baselines to evaluate decentralized aggregation and measure each system's performance of processing concurrent windows in a decentralized setup. We first measure the network overhead for concurrent window workloads and then measure the latency of different nodes in a single window workload.

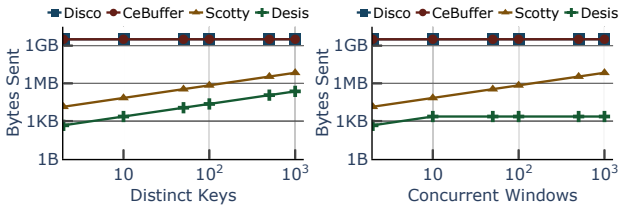
6.4.1 Network Overhead. We investigate the network overhead of all systems in a 3-node cluster. We send 100 million events to each system and calculate network overheads of local and intermediate nodes. We then gradually add more concurrent windows and distinct keys to measure the overall network overheads for each system.

Results. We first show the results for one query with an average function in Figure 11a. Desis and Disco have much lower network utilization than CeBuffer and Scotty. Centralized systems have to collect all events in their root node, while Desis and Disco only send partial results, which saves 99% of network



(a) Network overhead of average.

(b) Network overhead of median.



(c) Network overhead of distinct keys.

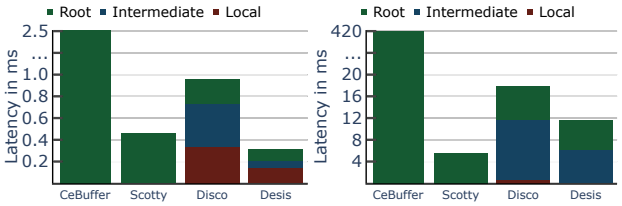
(d) Network overhead of concurrent windows.

Figure 11: Network overhead of systems by different nodes.

overhead. In the case of a non-decomposable median function (Figure 11b) all systems have to transfer the events to the root node. Desis, Scotty, and CeBuffer all have about 3GB network overhead, Disco is higher because it uses strings to send events and messages between nodes, while all other systems send bytes directly. Compared to non-decomposable functions, decomposable functions have fewer partial results and high reduction factors. In Figure 11c, we measure network overheads with a single query but increase the number of distinct keys. The network overhead of Desis and Disco increases linearly with more distinct window keys involved because partial results from different keys have to be calculated and transmitted individually. However, when we process concurrent windows with a single key the network overhead of Desis is constant (Figure 11d). This is due to the fact that Desis computes slices instead of queries in local nodes. Local nodes do not repeatedly send partial results of overlapping windows that can be shared.

We also observe that, when processing decomposable functions, the network overheads of local nodes and intermediate nodes in centralized systems (Scotty and CeBuffer) are the same. The intermediate nodes only transfer data to their upper layers. The network overhead will linearly increase in a complicated topology with multiple intermediate layers between local and root nodes. In decentralized systems (Desis and Disco), the increase in network overhead is negligible with respect to different network topologies because most network overhead of sending individual events is eliminated by partial results. For non-decomposable functions events have to be sent to parent nodes in any case, a complicated topology also leads to more network overheads in decentralized systems.

Summary. We conclude that decentralized aggregation outperforms centralized aggregation regarding saving network overhead when processing decomposable functions, especially in a complicated network that has multiple intermediate nodes between locals and the root. The decomposable functions have a higher reduction factor than non-decomposable functions. Additionally, Desis can save network overheads by sharing partial results between overlapping windows when processing concurrent windows.



(a) Latency of average.

(b) Latency of median.

Figure 12: Latency of systems by different nodes.

6.4.2 Latency. We keep the same setup as before and measure latency for local nodes, intermediate nodes, and root nodes. We measure latency by recording the time for systems performing window aggregations. Scotty and CeBuffer only process events on the root node and their root node latency is their system latency. All systems process one 1 second sized tumbling window.

Results. In Figure 12a we see that CeBuffer has the highest latency since it cannot perform incremental aggregation. All nodes in Desis and Disco contributed to the overall latency. Therefore, the network topology affects the latency of a decentralized system. In a topology that has multiple intermediate layers, the overall latency is increased because of multiple latencies in the intermediate nodes. In Figure 12b, we show the latency of query with median aggregation. We can see that the latency in the local nodes of Desis is much lower than that in intermediate nodes and the root node. This is because the local nodes transmit event batches to the intermediate nodes, and all batches are merged and processed there, which is more expensive than performing decomposable function aggregations.

Summary. We see that the latency of decentralized aggregation is affected by the network topology, and it increases linearly with the number of intermediate layers. In a minimal topology, Desis outperforms all other baselines when processing decomposable aggregations.

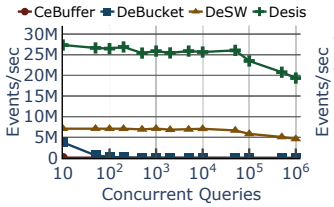
6.5 Real-World Performance

In this workload, we measure Desis and the baselines in a realistic setup. We let data generators read events from the DEBS 2013 dataset [46].

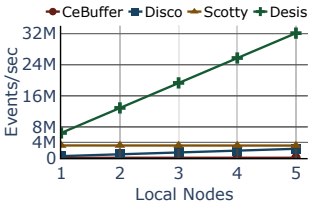
6.5.1 Real-World Data. We study the throughput of Desis, DeSW, DeBucket, and CeBuffer with real-world data and a large number of concurrent queries. We let query generators randomly produce queries with different keys, window types, window measures, decomposable functions, and window lengths.

Results. In Figure 13a, the throughput of CeBuffer and DeBucket decrease rapidly with the number of queries. Compare to DeSW, Desis has about 4 times better performance. When the number of queries is beyond 10K, the throughput of Desis and DeSW also decrease. This is because the root node has to generate the final results for each query. There all query results need to be materialized separately. When processing a large number of queries, the generation of the final results becomes dominant.

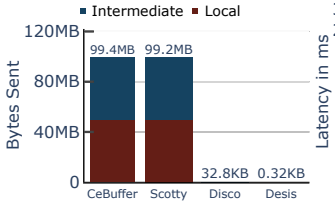
Summary. The throughput of Desis is constant even with a high number of parallel queries but decreases beyond 10K. This can be mitigated by separating queries to multiple root nodes.



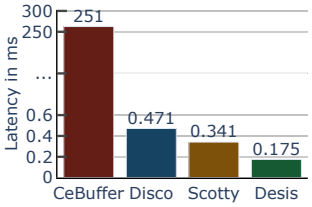
(a) Throughput on real-world data.



(b) Throughput on Raspberry Pi Cluster.



(c) Network overhead on Raspberry Pi Cluster.



(d) Latency on Raspberry Pi Cluster.

6.5.2 Performance on Raspberry Pi Cluster. We evaluate the performance of Desis, Disco, Scotty, and CeBuffer in a Raspberry Pi cluster with 1G Ethernet. Every Raspberry PI has one 4-Core Cortex-A72, 8 GB memory, and Debian 11 64-bit system. To distinguish with Raspberry Pi, we denote our node as Intel node. We use one Intel node as the root and two Raspberry Pis as intermediate and local nodes. Every system processes concurrent tumbling windows with average functions and 10 distinct keys. We first measure network overhead produced every second and latency of each system. We then gradually increase the number of Raspberry Pi nodes to measure throughput.

Results. In Figure 13c and 13d, we see that Scotty and CeBuffer transfer 99 MB per second, i.e., about 3.2 million events/s, which is also the maximum bandwidth of the Raspberry Pi cluster. Compare to Figure 6a, Disco and Desis have roughly the same latency when processing events on a Raspberry Pi node. In Figure 13b, Desis has 6.4 million events/s throughput and it scales linearly with the number of Raspberry Pi nodes. However, the throughput of Scotty stays at 3.2 million events/s even if we add more Raspberry Pis. Scotty's throughput is less than that on Intel node cluster because its performance is limited by network bandwidth.

Summary. Desis has low latency and high throughput and outperforms the baselines when processing concurrent windows on a Raspberry Pi cluster representing a distributed setup. The performance of the centralized aggregation is affected by low network bandwidth since fewer events can be sent to the root.

7 RELATED WORK

Stream processing engines. Recent work on scale-up systems, such as Saber [37], Streambox [45], BriskStream [68], Trill [16], StreamInsight [34], Grizzl [22]. They efficiently take advantage of modern hardware that can process events with high throughput, low latency, and exactly-once semantics. Further examples of scale-out systems are Flink [6, 14], Spark [64], Storm [57], Kafka Streams [51], Apache Beam [8] ,and MillWheel [2]. These approaches utilize a distributed processing model that executes computations on a cluster. All those systems are developed for the single query workload and non-decentralized networks. When processing multiple queries with different window types, aggregation functions, and window measures, they have to buffer all events and process each query individually. To process multiple queries in decentralized networks, they collect all data to their cluster and process queries there. However, our Desis can split concurrent windows into slices and push down these slices to nodes close to data streams and share partial results between different windows. So we can reduce redundant calculations and network overheads and move computation load from the center to all nodes in a decentralized network.

Window Aggregation. Current work on window aggregation [11, 19, 54, 67] focuses on different optimization of incremental aggregation. For multiple queries that have different window types and aggregation functions, those solutions have to process multiple queries individually. Pairs [38], Panes [39], LightSaber [56], and D-Stream [65] can share partial results between windows, but window types are limited to tumbling and sliding windows. Cutty [15] and Scotty [58–60] support arbitrary user-defined windows. However, they do not have techniques to optimize queries with different aggregation functions. Compared to previously mentioned solutions, Desis can share partial results between arbitrary windows regardless of window types, aggregation functions, and window measures.

Decentralized Aggregation. Madden et al. [44] introduce a tree-structured method that can perform partial aggregation, but it only supports tumbling windows and decomposable functions. Cougar [62], LEACH [25] and Directed Diffusion [27] are all based on tree-structured, and make efforts to create an efficient topology, but they all focus on tumbling windows. PEGASIS [42] presents a chain-structured approach, which is more expensive than tree-structured compared to network overhead. Daiki et al. [53] provide an approximate distributed aggregation algorithm that is only for simple decomposable functions. Disco [10] has the same structure as Desis, and can process complicated windows, but does not work well with multiple queries. In our work, Desis supports arbitrary windows and aggregation functions, and can efficiently process concurrent queries in decentralized networks.

8 CONCLUSION

In this paper, we present Desis, a stream processing system that can process multiple queries efficiently. For concurrent windows that have different window types, aggregation functions, and window measures, Desis split windows into slices and share partial results between them. Also, Desis supports decentralized aggregation that utilizes adaptive optimizations to deal with multiple queries in a decentralized network. In our evaluation, we make a fair comparison between Desis and state-of-art solutions and show a noticeable improvement in both throughput and network overheads. Furthermore, we explore the characteristics of different queries and figure out which one is beneficial for our system. The results reveal that Desis can consistently outperform baselines with scaling to a large number of queries.

ACKNOWLEDGMENTS

This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and the HPI research school on Data Science and Engineering.

REFERENCES

[1] Zainab Salih Ageed, Subhi RM Zeebaree, Mohammed Mohammed Sadeeq, Shakir Fattah Kak, Zryan Najat Rashid, Azar Abid Salih, and Wafaa M Abdullah. 2021. A survey of data mining implementation in smart city applications. *Qubahan Academic Journal* 1, 2 (2021), 91–99.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[3] 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] Tyler Akidau, Slava Chernyak, and Reuven Lax. 2018. *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.".

[5] Shahriar Akter and Samuel Fosso Wamba. 2016. Big data analytics in E-commerce: a systematic review and agenda for future research. *Electronic Markets* 26, 2 (2016), 173–194.

[6] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. 2014. The stratosphere platform for big data analytics. *The VLDB Journal* 23, 6 (2014), 939–964.

[7] Alibaba. 2020. *Four Billion Records per Second!* Alibaba. Retrieved Dec 2, 2020 from https://www.alibabacloud.com/blog/

[8] Apache Apex. 2018. Enterprise-grade unified stream and batch processing engine.

[9] Cagri Balkesen and Nesime Tatbul. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *International workshop on data management for sensor networks (DMSN)*.

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

[11] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. 2014. Slider: Incremental sliding window analytics. In *Proceedings of the 15th international middleware conference*. 61–72.

[12] Brice Bingman. 2018. Poor performance with sliding time windows. *Flink Jira Issues (issues. apache. org/jira/browse/FLINK-6990)* (2018).

[13] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. 2010. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 232–243.

[14] 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).

[15] 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.

[16] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[17] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

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

[19] Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. 2006. Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering* 19, 1 (2006), 57–72.

[20] Norjihan Abdul Ghani, Suraya Hamid, Ibrahim Abaker Targio Hashem, and Ejaz Ahmed. 2019. Social media big data analytics: A survey. *Computers in Human Behavior* 101 (2019), 417–428.

[21] 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, 1 (1997), 29–53.

[22] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2487–2503.

[23] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, and Alexandros Labrinidis. 2011. Optimized processing of multiple aggregate continuous queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 1515–1524.

[24] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, and Alexandros Labrinidis. 2012. Three-level processing of multiple aggregate continuous queries. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 929–940.

[25] 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.

[26] Ayae Ichinose, Atsuko Takefusa, Hidemoto Nakada, and Masato Oguchi. 2017. A study of a video analysis framework using Kafka and spark streaming. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2396–2401.

[27] 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.

[28] 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.

[29] Mohd Javaid, Abid Haleem, Ravi Pratap Singh, and Rajiv Suman. 2021. Significant applications of big data in Industry 4.0. *Journal of Industrial Integration and Management* 6, 04 (2021), 429–447.

[30] 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.

[31] 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.

[32] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: ad-hoc stream joins at scale. *Proceedings of the VLDB Endowment* 13, 4 (2019), 435–448.

[33] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data*. 607–622.

[34] Seyed Jalal Kazemitabar, Ugur Demiryurek, Mohamed Ali, Afsin Akdogan, and Cyrus Shahabi. 2010. Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1537–1540.

[35] 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.

[36] Jyoti Mante Khurpade, Devakanta Rao, and Parth D Sanghavi. 2018. A Survey on IOT and 5G Network. In *2018 International conference on smart city and emerging technology (ICSCET)*. IEEE, 1–3.

[37] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*. 555–569.

[38] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 623–634.

[39] 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.

[40] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 311–322.

[41] Jin Li, Kristin Tufte, David Maier, and Vassilis Papadimos. 2008. AdaptWID: An adaptive, memory-efficient window aggregation implementation. *IEEE Internet Computing* 12, 6 (2008), 22–29.

[42] Stephanie Lindsey, Cauligi Raghavendra, and Krishna M Sivalingam. 2002. Data gathering algorithms in sensor networks using energy metrics. *IEEE Transactions on parallel and distributed systems* 13, 9 (2002), 924–935.

[43] Guojin Liu, Rui Tan, Ruogu Zhou, Guoliang Xing, Wen-Zhan Song, and Jonathan M Lees. 2013. Volcanic earthquake timing using wireless sensor networks. In *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 91–102.

[44] 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)*.

[45] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2017. {StreamBox}: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 617–629.

[46] 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.

[47] Snehal Nagmote and Pallavi Phadnis. 2019. Massive scale data processing at netflix using flink. In *Flink Forward Conference*.

[48] Pekka Pääkkönen. 2016. Feasibility analysis of AsterixDB and Spark streaming with Cassandra for stream-based processing. *Journal of Big Data* 3, 1 (2016), 1–25.

[49] 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.

[50] Radhya Sahal, John G Breslin, and Muhammad Intizar Ali. 2020. Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case. *Journal of manufacturing systems* 54 (2020), 138–151.

[51] 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.

[52] Leo Syinchwun. 2016. Lightweight event time window. *Flink Jira Issues (issues. apache. org/jira/browse/FLINK-5387)* (2016).

[53] Daiki Takao, Kento Sugiura, and Yoshiharu Ishikawa. 2021. Approximate Fault-Tolerant Data Stream Aggregation for Edge Computing. In *International Conference on Big Data Analytics*. Springer, 233–244.

[54] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment* 8, 7 (2015), 702–713.

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

[56] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient window aggregation on multi-core processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2505–2521.

[57] 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.

[58] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2018. Scotty: Efficient window aggregation for out-of-order stream processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1300–1303.

[59] Jonas Traub, Philipp M Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing.. In *EDBT*, Vol. 19. 97–108.

[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] 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.

[63] Shen Yin and Okyay Kaynak. 2015. Big data for modern industry: challenges and trends [point of view]. *Proc. IEEE* 103, 2 (2015), 143–146.

[64] 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.

[65] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and {Fault-Tolerant} Model for Stream Processing on Large Clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.

[66] 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).

[67] Chao Zhang, Reza Akbarinia, and Farouk Toumani. 2021. Efficient Incremental Computation of Aggregations over Sliding Windows. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2136–2144.

[68] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 International Conference on Management of Data*. 705–722.