In-Network Management of Parallel Data Streams over Programmable Data Planes

Bochra Boughzala
University of Groningen
Groningen, The Netherlands
b.boughzala@rug.nl

Boris Koldehofe
Technische Universität Ilmenau
Ilmenau, Germany
boris.koldehofe@tu-ilmenau.de

***Abstract*—Current data centers host an ever increasing number of data analytics applications who are dealing with a growing number of data sources and a continuously increasing volume of data. Parallel stream processing is a powerful paradigm supporting the large-scale deployment of data-analytics applications. However, its performance is limited by its processing capacity of splitting the data streams into parallelizable substreams. The splitter that is traditionally executed on general-purpose computational resources can benefit from in-network computing nodes on the communication path. Programmable data planes and corresponding programming models, e.g., Programming Protocol-independent Packet Processors (P4), offer the flexibility of enabling distinct parallelization semantics that can be individually adapted to the dynamic workload. In this paper, we propose Stateful and Scalable Splitter Switch (S4), a network-centric approach leveraging P4 to support parallel stream processing. S4 supports up to $286k$ concurrent data streams, with a parallelism degree of up to $\sim 500k$ operator instances and a latency overhead of only $2\mu s$.**

***Index Terms*—In-Network Computing, P4, Parallel Stream Processing.**

I. Introduction

Data analytics applications are becoming highly important in current data centers as they enable the processing of high volume of data coming from multiple data sources, e.g., Internet of Things (IoT) devices. With the growing number of data sources, it becomes increasingly challenging to ingest the rising volume of data, e.g., sensor measurements, training data for Machine Learning (ML) models. In this context, parallel stream processing is an important paradigm for enabling large scale deployments of data analytics applications. Particularly, data parallelism [1] is a very prominent mechanism in stream processing engines such as Apache Flink [2]. It is commonly achieved by the *splitter-merger* model [16] where the splitter divides the incoming event streams into parallelizable substreams called *windows*. Then, the splitter distributes the windows of events over a set of operator instances. A window is a finite set of events and it represents a fundamental building block in stream processing.

The efficient splitting of the various data streams into windows is crucial for the overall performance of the data-analytics application, especially when the computation results are expected in real-time. As the workload increases, it is essential that the splitter manages a higher ingestion rate when the data streams arrive at line rate. Moreover, current data centers host multiple data-analytics applications, each with its own parallelization needs. Therefore, scaling to a large number of concurrent data streams with specialized windowing semantics is key. It is also important to adapt the *parallelism degree* for supporting more operator instances without redeploying the entire system. However, increasing the number of operator instances is pointless when the splitter performance is the *bottleneck*. In fact, state-of-the-art stream processing engines implement the splitter on general-purpose computational resources where the splitter throughput for a given data analytics application is around $800k$ events per second with a parallelism of 256 operator instances [3].

Instead, network specialized resources in today's data centers can support the splitter to meet the requirements of enabling distinct window-based parallelization semantics and scaling them individually. For this purpose, we propose using Software-Defined Networking (SDN) and In-Network Computing (INC) [23] as it has the potential to accelerate the performance of the splitter as a network function. With emerging programmable data planes, *in-network* nodes can be specifically programmed for customized functions, making them appealing targets to experiment with novel use cases. For instance, since the advent of the P4 programming language [4], there have been tremendous innovations with new protocols and various domains of in-network applications, e.g., consensus [5], machine learning [6] and caching [7]. An important motivation for in-network computing is to reduce end-to-end latency by shorter communication paths, save network bandwidth [24], and benefit from hardware performance. While P4 programmable switches bring high flexibility for defining new packet headers and defining dependencies in packet processing, the programming model has limitations in expressing in-network computations such as loops and floating-point operations. Particularly, stateful processing in P4 is challenging [22] but it is essential for the splitter to support distinct parallelization semantics of multiple data analytics applications.

Building on in-network computing, we propose S4, a network-centric approach leveraging P4 to support parallel stream processing over programmable data planes. We show with S4 how P4 programmable switch can accelerate the splitter function while supporting distinct windowing semantics. Our main contributions are the following; (1) The proposi-

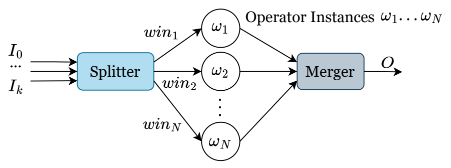


Figure 1: Data Parallelization Framework

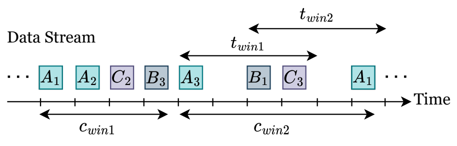


Figure 2: Count-based Tumbling Window ($n = 4$, $\delta = 4$) and Time-based Sliding Window ($n = 4$ sec, $\delta = 2$ sec).

tion of in-network data parallelization framework supporting window-based parallel data streams. (2) The development of data plane algorithms of four essential windowing semantics; count-based and time-based windows both in the tumbling, i.e., non-overlapping and sliding, i.e., overlapping mode. (3) An in-depth analysis of S4 properties in terms of load-balancing and hardware resources consumption depending on the window semantics to determine the highest parallelism degree achievable by S4. (4) A performance evaluation of S4 comprising the analysis of the throughput and measurements of the latency and the control plane overhead.

II. Parallel Stream Processing

Parallel stream processing is often realized by a data parallelization framework which relies on the splitter-merger architecture [16] (Figure 1). First, the splitter receives the incoming data streams $I_0 ... I_k$ where each stream comprises an infinite set of events originating from various data sources. The events usually have timestamps indicating their occurrence time. The splitter divides the data streams according to a splitting strategy employing key-based or window-based methods [19]. Particularly, window-based processing is essential for managing large datasets in real-time [11]. Then the windows, i.e., a finite set of events, are assigned by the splitter to a set of operator instances $\omega_1, \omega_2, ... \omega_N$, according to a scheduling policy. The operator instances are responsible for the parallel execution of local computations over the data partitions, i.e., windows. Lastly, the merger collects and aggregates the computation results into an output stream O.

A. Window-based Data Stream Partitioning

In stream processing, windowing represents a powerful abstraction for enabling the parallel execution of replicated computations over a subset of the data stream. Window-based data partitioning can be realized according to different *windowing semantics*. Given the window specification Σ_i which comprises the window size n_i and window shift δ_i for a particular input stream, the splitter partitions the incoming data stream into windows of events, such as a window win_j is a finite set of n tuples of timestamped events $(e_0, t_0), (e_1, t_1) ... (e_{n-1}, t_{n-1})$. There are time-based and count-based windows both of which can be defined according to the *tumbling* windows (non-overlapping) or the *sliding* windows (overlapping) model (Figure 2). An example of windowed operation is a traffic sensor that needs to count the number of vehicles passing by a particular location every minute.

B. Window Scheduling and Load-Balancing

Alongside the windowing operation, the splitter performs the load-balancing of the windows of events among the set of operator instances. It assigns the windows according to a scheduling mechanism such as Round Robin (RR) or Weighted Round Robin (WRR). The number of operator instances managed by the splitter represents the *parallelism degree* N and is a key indicator of the splitter performance and the scaling capacity of the streaming system. As the workload increases, additional operator instances may be deployed to increase the system throughput and reduce its latency. Therefore, having an adaptive splitter that allows the dynamic reconfiguration of the parallelism degree N is an essential property.

III. Design Overview and System Requirements

In this section, we give the design overview of the proposed in-network data parallelization framework and describe corresponding system requirements.

A. Design Overview

Our proposed in-network data parallelization framework is presented in Figure 3. Multiple data analytics applications are assigned to distinct input streams. A P4 programmable switch called S4 (Stateful and Scalable Splitter Switch) receives the incoming streams on its ingress ports and executes the line-rate forwarding of events while performing the splitting function (windowing and load-balancing). S4 enables the individual configurations of the distinct application streams allowing for each of the data analytics applications independent window semantics. Each input stream I_i is associated with a custom windowing semantic expressed by the window specification Σ_i. In S4, an event e_i corresponds to a packet carrying a timestamp t_i of the event occurrence time. Events belonging to the same application have the same event type, e.g., temperature, humidity. Similar to the traditional model, the operator instances are processes executed on general-purpose computational resources, i.e., servers. The data parallelization configurations (window specification Σ and parallelism degree N) for each installed stream and associated operator instances are provided to the S4 switch by the SDN controller via the control plane interface. The stream processing engine managing the various data analytics applications monitors the dynamic workload and the state of the operator instances to adjust the parallelism degree accordingly in the form of updated configuration.

Bochra Boughzala, and Boris Koldehofe. **In-Network Management of Parallel Data Streams over Programmable Data Planes**. In Proceedings of the 23rd International Federation for Information Processing (IFIP) Networking 2024 Conference (NETWORKING 2024), IFIP, 9 pages, 2024.

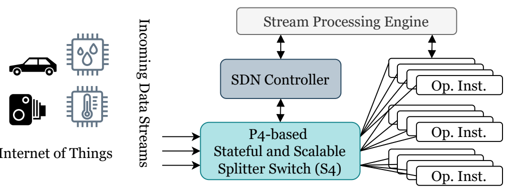


Figure 3: In-network Parallel Stream Processing with P4 Programmable Switch.

B. System Requirements

S4 enables line-rate decisions for each incoming packet to be mapped to a specific set of windows and then forwards it to one or multiple operator instances. With tumbling windows, an event belongs to a single window and is forwarded to a single operator. While in the case of sliding windows an event belongs to multiple simultaneously active windows and must be forwarded to corresponding sets of parallel operators. As event packets are received, S4 updates the window state dependent on the concrete window semantic. S4 must ensure a consistent partitioning and mapping of the data stream such that all events in a given window win_j are consistently assigned to the same operator instance ω_k. A key challenge for the proposed approach results from the required state management with respect to each of the configured streams. Therefore, S4 builds on the P4 pipeline resources, i.e., Match-Action Tables (MAT), Packet Replication Engine (PRE) and particularly those supporting stateful processing i.e., registers, to process incoming events and update the progress of active windows related to each individual stream. Moreover, to adapt to the dynamic workload S4 must enable the dynamic changes to the configured streams and their parallelization degree.

In the remainder of the paper, we aim to determine concrete algorithms that ensure consistent updates of the following four central windowing semantics, Count-Based Tumbling Window (CBTW), Count-Based Sliding Window (CBSW), Time-Based Tumbling Window (TBTW) and Time-Based Sliding Window (TBSW).

IV. Stateful and Scalable Splitter Switch

In this section, first we describe the data plane pipeline processing of S4. Then, we present the four data plane algorithms of the window-based data stream partitioning.

A. S4 Data Plane Pipeline Processing

An overview of S4 pipeline processing is presented in Figure 4. An input data stream is expected to arrive at a given ingress port of the P4 switch. We use the ingress port ID along with the event type in a MAT lookup to assign a Stream ID to the event upon its reception. In the same MAT, the window type, i.e., count-based or time-based, and the window specification parameters, e.g., $\Sigma(n, \delta)$ for count-based windows or $\Sigma(\Delta, \tau)$ for time-based windows, are also set as action data and later used as user metadata in the remainder of the pipeline. A following MAT lookup uses the Stream ID as a key to retrieve the parallelism degree N which is

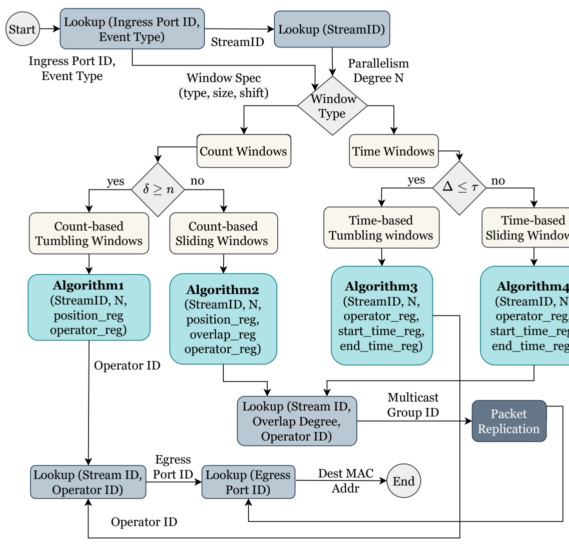

Start
Lookup (Ingress Port ID, Event Type)
StreamID
Lookup (StreamID)
Ingress Port ID, Event Type
Window Spec (type, size, shift)
Parallelism Degree N
Window Type
Count Windows
Time Windows
yes
$\delta \geq n$
no
yes
$\Delta \leq \tau$
no
Count-based Tumbling Windows
Count-based Sliding Windows
Time-based Tumbling windows
Time-based Sliding Windows
Algorithm1 (StreamID, N, position_reg operator_reg)
Algorithm2 (StreamID, N, position_reg, overlap_reg operator_reg)
Algorithm3 (StreamID, N, operator_reg, start_time_reg, end_time_reg)
Algorithm4 (StreamID, N, operator_reg, start_time_reg, end_time_reg)
Operator ID
Lookup (Stream ID, Overlap Degree, Operator ID)
Multicast Group ID
Packet Replication
Lookup (Stream ID, Operator ID)
Egress Port ID
Lookup (Egress Port ID)
Dest MAC Addr
End
Operator ID

Figure 4: S4 Data Plane Pipeline Processing

done separately from the window specification so that for each input stream both can be updated independently. The window specification parameters and the parallelism degree are defined as action data configurable by the control plane so they can be updated dynamically at run-time. Hence, an update to the window specification Σ or the parallelism degree N does not require a recompilation of the P4 program. Once the window parameters are retrieved, S4 pipeline processing enters one of the following control blocks: Algorithm 1 (CBTW), Algorithm 2 (CBSW), Algorithm 3 (TBTW), and Algorithm 4 (TBSW).

Performing the window-based data stream splitting logic in the data plane requires keeping the state of the windows and their mapping to the destination operators throughout the window lifetime for non-overlapping windows and the state of multiple concurrent windows to a set of operator instances for sliding windows. For this purpose, we rely on the use of registers in P4 as they can be modified from within the data plane and the stored values are persisted beyond the life of a packet. We use the Stream ID as the *index* to access the various register elements associated with a particular data stream. Thus, the state management is done in isolation for each of the configured streams. In the case of non-overlapping windows the output of corresponding processing blocks (Algorithm 1 and 3) is an Operator ID which is then mapped to an egress port ID. In the case of overlapping windows (Algorithm 2 and 4), an event can be part of multiple simultaneously active windows, each of which is mapped to a different operator instance. Therefore, packet replication is required so that the event packet is multicast towards respective operators. The output of associated processing block is a Multicast Group ID with a configuration comprising the output ports mapped to the set of operator instances involved with the ongoing active windows. The pipeline processing finishes by updating the source and destination Media Access Control Address (MAC) addresses based on the destination operator and the switch egress port.

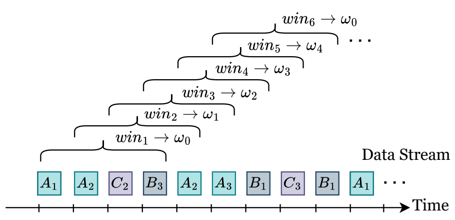
B. S4 Window-based Data Stream Partitioning

*1) **Count-based Tumbling Windows (Algorithm 1):*** With non-overlapping count-based windows, we have $n \leq \delta$. We define two registers arrays (i) *position_reg* for tracking the total number of received packets within the current window. (ii) *operator_reg* for keeping the latest operator ID, to which the currently active window is assigned. When an event is received, the *position_reg* element identified by the Stream ID is incremented by 1. Then, a check if a new window has started would trigger advancing the *operator_reg* by 1 until it reaches the maximum number of operator instances N in which case, it is reset to 0. Otherwise, if the event is part of the current window, then the Operator ID remains the same as the previous packet, and its value is retrieved from the *operator_reg* register. A MAT lookup is then performed based on the operator ID and the Stream ID to retrieve the egress port ID.

Algorithm 1 Count-based Tumbling Windows

```
Require: Stream ID $idx$, parallelism degree $N$, window size $n$,
    window shift $\delta$, $n \leq \delta$.
Ensure: Setting the egress port ID based on the Operator ID.
procedure POSITIONREGACTION($positionReg, idx$)
    if ($value < \delta - 1$) then
        $value \leftarrow value + 1$; $flag \leftarrow 0$;
    else
        $value \leftarrow 0$; $flag \leftarrow 1$;
    end if
    return $flag$;
end procedure
procedure OPERATORREGACTION($operatorReg, idx$)
    if ($value < N - 1$) then
        $value \leftarrow value + 1$;
    else
        $value \leftarrow 0$;
    end if
    return $value$;
end procedure
$n, \delta, idx \leftarrow windowSpecTable(igPort, hdr.event.type)$
$N \leftarrow streamIDtoMaxOperatorNumTable(idx)$
$newWindow \leftarrow positionRegAction(idx)$
if $newWindow == 1$ then
    $\omega \leftarrow operatorRegAction(idx)$
else
    $\omega \leftarrow operatorReg.read(idx)$
    $pos \leftarrow positionReg.read(idx)$
end if
$portID \leftarrow operatorIDtoPortIDmappingTable(idx, \omega)$
```

*2) **Count-based Sliding Windows (Algorithm 2):*** With overlapping count-based windows ($n > \delta$), we need an additional register *overlap_reg* for tracking the overlap degree among the currently active windows. We also define a MAT *slidingWindows* to set the mapping of the overlapping windows to the corresponding set of operator instances. To understand sliding windows from the data plane point of view let's consider the case when $\delta = 1$. Then for every incoming event, a new window is created and assigned to the next operator (assuming a round-robin scheduling). Given the example illustrated in Figure 5 where Stream ID $0x123$ is partitioned according to $n = 4, \delta = 1$ and distributed among $N = 5$. For the first received event, there is no overlap and the packet must be sent only to the first operator ω_0. However, starting from the second event, the window overlap begins and the event must be sent to operator ω_0 and ω_1, then the next packet is sent to operator ω_0, ω_1 and ω_2 and so on as described in Table I. To express this behaviour, the *overlap_reg* is used for tracking the current overlap degree. In the given example, the overlap degree starts at 0, then it is incremented by 1 for each incoming packet until it reaches a maximum overlap value $maxOverlap = n - \delta$. We call this process the *start-up* phase. Then, we use the overlap degree along with the stream ID and the latest operator ID as a lookup key in the *slidingWindows* MAT to apply one of these actions; (1) *unicast forwarding* by setting the egress port ID so the event packet is sent to a single operator (first entry in Table I) (2) *multicast forwarding* for replicating the event across multiple operators (the rest of the entries in Table I). Packet replication is ensured by the traffic manager in the P4 switch, given the configuration of the multicast tree (multicast group IDs and associated port members) in accordance with the entries in the *slidingWindows* table (see example in Table I).



$win_6 \rightarrow \omega_0$
$win_5 \rightarrow \omega_4$
$win_4 \rightarrow \omega_3$
$win_3 \rightarrow \omega_2$
$win_2 \rightarrow \omega_1$
$win_1 \rightarrow \omega_0$
Data Stream
A_1 A_2 C_2 B_3 A_2 A_3 B_1 C_3 B_1 A_1 ...
Time

Figure 5: Count-based Sliding Windows with $\Sigma(n = 4, \delta = 1)$ and $N = 5$.

idx	latest Op.	$Overlap$	Op. Inst.	Mcast Group ID
$0x123$	ω_0	0	ω_0	-
$0x123$	ω_1	1	ω_0, ω_1	1
$0x123$	ω_2	2	$\omega_0, \omega_1, \omega_2$	2
$0x123$	ω_3	3	$\omega_0, \omega_1, \omega_2, \omega_3$	3
$0x123$	ω_4	3	$\omega_1, \omega_2, \omega_3, \omega_4$	4
$0x123$	ω_0	3	$\omega_2, \omega_3, \omega_4, \omega_0$	5
$0x123$	ω_1	3	$\omega_3, \omega_4, \omega_0, \omega_1$	6
$0x123$	ω_2	3	$\omega_4, \omega_0, \omega_1, \omega_2$	7
$0x123$	ω_3	3	$\omega_0, \omega_1, \omega_2, \omega_3$	3

Table I: Round-Robin window load-balancing over 5 operator instances with $\Sigma(n = 4, \delta = 1)$.

For the general case of count-based sliding windows when $\delta > 1$, an additional register is needed for tracking the start of a new window. We can use the register *position_reg* so when it reaches the value δ it would trigger the start of a new window. Then, the other two registers *operator_reg* and *overlap_reg* will also be advanced.

*3) **Time-based Tumbling Windows (Algorithm 3):*** For non-overlapping time-based windows, a window specification is given by the window duration Δ and the window shift τ as

Algorithm 2 Count-based Sliding Windows

```
Require: Stream ID $idx$, parallelism degree $N$, window size $n$,
    window shift $\delta$, $overlap$ degree, $n > \delta$.
Ensure: Setting the egress $portID$ or Multicast group ID $mgID$.
procedure OVERLAPREGACTION($overlapReg, idx$)
    if ($value < maxOverlap$) then
        $value \leftarrow value + 1$;
    end if
    return $value$;
end procedure
$n, \delta, idx \leftarrow windowSpecTable(igPort, hdr.event.type)$
$N \leftarrow streamIDtoMaxOperatorNumTable(idx)$
$maxOverlap \leftarrow n - \delta$
if $\delta == 1$ then
    $\omega \leftarrow operatorRegAction(idx)$
    $overlap \leftarrow overlapRegAction(idx)$
end if
if $\delta > 1$ then
    $newWindow \leftarrow positionRegAction(idx)$;
    if $newWindow == 1$ then
        $\omega \leftarrow operatorRegAction(idx)$;
        $overlap \leftarrow overlapRegAction(idx)$;
    else
        $\omega \leftarrow operatorReg.read(idx)$;
        $overlap \leftarrow overlapReg.read(idx)$;
    end if
end if
$mgID, portID \leftarrow slidingWindows(idx, overlap, \omega)$;
```

measures of time such as $\Delta \leq \tau$. For each received event, S4 starts by parsing the event packet header to retrieve the event timestamp. Assuming an in-order arrival of the events within a given stream, the event timestamps are discrete and continuous monotonically increasing integers. For a new window triggered by the arrival of an event packet with a timestamp t_i, first we compute and save the current window end time as $curr\_w\_end\_time = t_i + \Delta$ in the register *end_time_reg*, and the next window starting time as $next\_w\_start\_time = t_i + \tau$ in the register *start_time_reg*. Note that if $\Delta = \tau$, then $curr\_w\_end\_time = next\_w\_start\_time$. Similar to count-based windows, we use the *operator_reg* register for keeping the state of the current operator ID. However, unlike count-based windows, we do not need the *position_reg*, since the advancement steps are given by the event timestamps rather than the packet count. When $t_i \geq next\_w\_start\_time$, then a new window is created so we start sending to the next operator.

*4) **Time-based Sliding Windows (Algorithm 4)**:* Overlapping time-based windows are created when $\Delta > \tau$. Same as with count-based overlapping windows, we use the *overlap_reg* to keep track of the current overlap degree. We define the maximum overlap degree for time-based sliding windows $maxOverlap = \frac{\Delta}{\tau}$. Since there are limitations to floating point operations in programmable switches, the maximum overlap degree parameter is given as action data to be configured along with the window specification and passed via the control plane interface. The rest of the processing of how the overlap degree is used is similar to count-based sliding windows. We also maintain the *slidingWindows* MAT where the Stream ID, the latest Operator ID and the current

Algorithm 3 Time-based Tumbling Windows

```
Require: Stream ID $idx$, window duration $\Delta$, window shift $\tau$,
    parallelism degree $N$, $\Delta \leq \tau$, $hdr.event.timestamp$.
Ensure: Setting the egress port ID based on the Operator ID.
procedure ENDTIMEREGACTION($endTimeReg, idx$)
    if ($hdr.event.timestamp >= value$) then
        $value \leftarrow currWinEndTime$; $flag \leftarrow 1$;
    else
        $flag \leftarrow 0$;
    end if
    return $flag$;
end procedure
procedure STARTTIMEREGACTION($startTimeReg, idx$)
    if ($hdr.event.timestamp >= value$) then
        $value \leftarrow nextWinStartTime$; $flag \leftarrow 1$;
    else
        $flag \leftarrow 0$;
    end if
    return $value$;
end procedure
procedure CURRSTARTTIMEREGACTION($startTimeReg, idx$)
    $value \leftarrow hdr.event.timestamp$;
    return $value$;
end procedure
$\Delta, \tau, idx \leftarrow windowSpecTable(igPort, hdr.event.type)$;
$N \leftarrow streamIDtoMaxOperatorNumTable(idx)$;
$currWinEndTime \leftarrow hdr.event.timestamp + \Delta$;
$nextWinStartTime \leftarrow hdr.event.timestamp + \tau$;
$windowStops \leftarrow endTimeRegAction(idx)$;
$newWindow \leftarrow startTimeRegAction(idx)$;
if $newWindow == 1$ then
    $\omega \leftarrow operatorRegAction(idx)$;
    $currStartTimeRegAction(idx)$;
else
    $\omega \leftarrow operatorReg.read(idx)$;
    $currWinStartTime \leftarrow startTimeReg.read(idx)$;
end if
$portID \leftarrow operatorIDtoPortIDmappingTable(idx, \omega)$
```

overlap degree are used as the lookup key to determine the outcome action (i) *unicast* forwarding in case of no-overlap (ii) *multicast* forwarding when there is overlap.

V. Evaluation

In this section, first we analyze the behaviour of S4 in terms of load distribution (Section V-B1). Then, we study the hardware resource consumption of S4 on a physical switch, the Intel Tofino1 switch [8], to identify the highest achievable parallelism degree and the maximum number of concurrent data streams for each of the windowing semantics (Section V-B2). Then, we present the performance measurements of S4 (Section V-B3) in terms of throughput, the latency overhead and the control plane overhead.

A. Evaluation Setup

The hardware testbed used in S4 evaluation comprises two Intel Tofino1 switches [8], an APS Networks 8x100Gbps - 48x10 Gbps switch and an EdgeCore 32x100 Gbps switch.

In the first Tofino switch, we deploy the S4 program which we implemented using $P4_{16}$ programming language

Bochra Boughzala, and Boris Koldehofe. **In-Network Management of Parallel Data Streams over Programmable Data Planes**. In Proceedings of the 23rd International Federation for Information Processing (IFIP) Networking 2024 Conference (NETWORKING 2024), IFIP, 9 pages, 2024.

Algorithm 4 Time-based Sliding Windows

```
Require: Stream ID $idx$, window duration $\Delta$, window shift $\tau$, parallelism degree $N$, overlap degree, $\Delta > \tau$, $hdr.event.timestamp$.
Ensure: Setting the egress $portID$ or Multicast group ID $mgID$.
1  $\Delta, \tau, idx \leftarrow windowSpecTable(igPort, hdr.event.type)$;
2  $N \leftarrow streamIDtoMaxOperatorNumTable(idx)$;
3  $maxOverlap \leftarrow \frac{\Delta}{\tau}$
4  $currWinEndTime \leftarrow hdr.event.timestamp + \Delta$;
5  $nextWinStartTime \leftarrow hdr.event.timestamp + \tau$;
6  $newWindow \leftarrow startTimeRegAction(idx)$;
7  $windowStops \leftarrow endTimeRegAction(idx)$;
8  if $newWindow == 1$ then
9      $\omega \leftarrow operatorRegAction(idx)$;
10     $overlap \leftarrow overlapRegAction(idx)$;
11     $currStartTimeRegAction(idx)$;
12 else
13     $\omega \leftarrow operatorReg.read(idx)$;
14     $overlap \leftarrow overlapReg.read(idx)$;
15     $currWinStartTime \leftarrow startTimeReg.read(idx)$;
16 end if
17 $mgID, portID \leftarrow slidingWindows(idx, overlap, \omega)$;
```

[9]. The second Tofino switch is only needed for a subset of the performance evaluation when we measure S4 latency overhead. In addition to the Tofino switches, our testbed comprises three additional rack servers: two 6-cores Intel Xeon Processor E5-2620 v2 servers which have a 82599ES 10-Gigabit SFI/SFP+ Intel NIC each and a 10-cores Intel Xeon Gold 5115 Processor on which we installed two 25-Gigabit Dual-Port SFP28 Intel E810-XXVAM2-Based Ethernet NICs. All links are configured at 10 Gbps speed. For traffic generation we use T-Rex [12], a Data Plane Development Kit (DPDK)-based [13] Realistic Traffic Generator.

B. Evaluation Results

*1) **Load Distribution**:* We define different testing scenarios where we aim to compare the four windowing semantics in terms of load-balancing. For each window model, we consider two configurations with different window specification Σ and parallelism degree N. Our goal is to determine whether the window parameters n and δ, or the number of operator instances N, have an influence on the load distribution. In each scenario, we instantiate S4 with one of the window specifications presented in Table II and Table III. We run the traffic generator T-Rex at full line speed on a 10GbE port. However, the incoming rate measured on the switch ingress port is 8.42 Gbps. The incoming data stream consists of UDP packets with a size of 128 Bytes. The timestamps are generated in-order as monotonically increasing integers.

The results presented in Figure 6 show the load distribution over the operator instances for the various window specifications. It shows that regardless of the window specification Σ and the parallelism degree N, our S4 implementation provides even load distribution at line-rate, in the case of overlapping and non-overlapping windows. We note that the bandwidth consumption of the sliding windows model is much higher than those of tumbling windows. Despite the communication

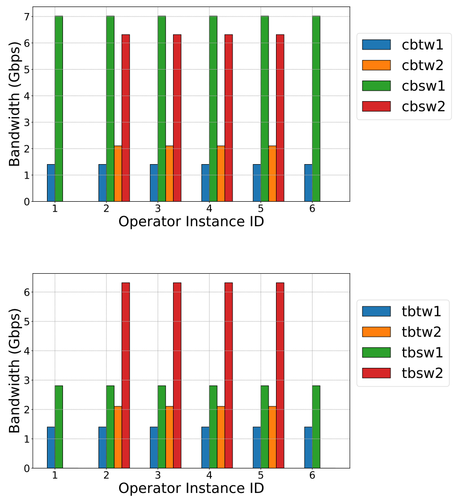


Figure 6: Load-Distribution over the Operator Instances for different Window Specifications

Table II: Count-based Window Specifications.

Scenario	Window Specification	N
cbtw1	$\Sigma(n = 100, \delta = 100)$	$N = 6$
cbtw2	$\Sigma(n = 10, \delta = 10)$	$N = 4$
cbsw1	$\Sigma(n = 5, \delta = 1)$	$N = 6$
cbsw2	$\Sigma(n = 3, \delta = 1)$	$N = 4$

overhead, sliding windows are essential in use cases defined by the data analytics application. For example, an application that computes every day the number of failures of hard drives in a data center over the past 30 days would implement the sliding windows operation [10].

*2) **Hardware Resources Consumption**:* To understand the hardware resource usage of our S4 implementation, first we define a baseline configuration where we set the parallelism degree $N = 64k$ operator instances, and the number of concurrent data streams to $32k$. First, we use the same baseline configuration for the different windowing semantics and we evaluate separately their hardware resources usage. The estimation of the hardware resources utilization is done with P4 Insight (*p4i*), a tool included in the Intel Tofino Software Development Environment (SDE) which provides the mapping of the P4 program to the switch hardware resources. Therefore,

Table III: Time-based Window Specifications.

Scenario	Window Specification	N
tbtw1	$\Sigma(\Delta = 1000, \tau = 1000)$	$N = 6$
tbtw2	$\Sigma(\Delta = 2000, \tau = 2000)$	$N = 4$
tbsw1	$\Sigma(\Delta = 1000, \tau = 500)$	$N = 6$
tbsw2	$\Sigma(\Delta = 1500, \tau = 500)$	$N = 4$

Bochra Boughzala, and Boris Koldehofe. **In-Network Management of Parallel Data Streams over Programmable Data Planes**. In Proceedings of the 23rd International Federation for Information Processing (IFIP) Networking 2024 Conference (NETWORKING 2024), IFIP, 9 pages, 2024.

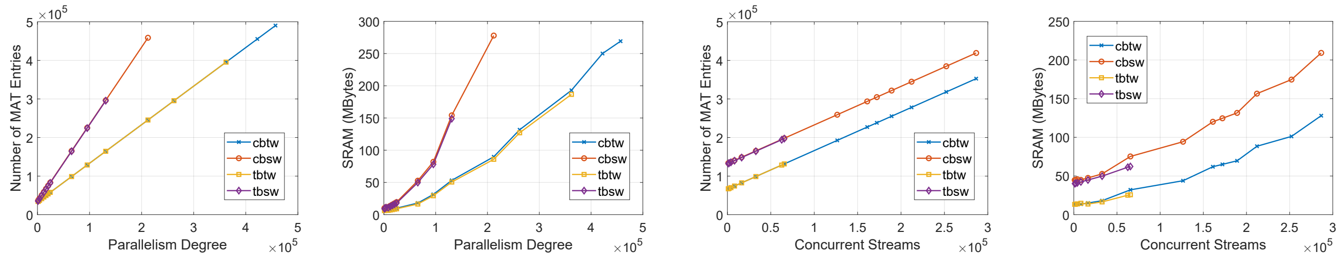


Figure 7: Hardware Resources Consumption in the S4 Switch Application.

Table IV: S4 Resource Utilization using the Baseline Configuration.

Resources	`cbtw`	`cbsw`	`tbtw`	`tbsw`
Registers Arrays	2	3	6	7
Match Tables	12	12	18	20
Stages	6	6	7	8

the resource utilization analysis can be performed offline. For the baseline configuration, we focus on the following essential hardware resources as comparative indicators of the hardware requirements for each of the windowing models (1) The number of register arrays. (2) The number of MAT tables. (3) The number of stages in the P4 pipeline.

The hardware resource utilization of our P4-based S4 implementation of each of the windowing semantics using the baseline configuration is described in Table IV. Note that Tofino1 has a maximum of 12 stages, beyond which the P4 program would not fit inside the switch. As shown in Table IV, with the baseline configuration we are below that limit in the four cases allowing the scaling of S4 beyond $64k$ operator instances and $32k$ concurrent streams. As shown in Table IV, both count-based tumbling and sliding windows use the least number of stages of the four windowing semantics as they both use 12 MAT tables over 6 stages. However, count-based tumbling windows require an additional register array. Time-based sliding windows have the longest dependency chain with 20 MAT tables spread over 8 stages. It also requires more register arrays (7 register arrays are needed in our implementation).

In the following part of our analysis, we study how scaling S4 for larger configurations impacts the hardware resources utilization inside the switch. Note that our evaluation of S4 hardware resources consumption is performed in stand-alone with no other networking functions sharing the same switch. The purpose of this study is to identify for each of the window models, the upper bound of the parallelism degree of S4 and how many data streams it can handle concurrently. We can then identify which of the proposed windowing semantics provides the best scalability.

For each window model, we start by fixing the number of concurrent streams to $32k$ and we vary the parallelism degree to a higher number of operator instances until all the stages of the Tofino switch are used. Then, for a fixed number of operator instances $N = 64k$, we vary the number of concurrent streams to the highest possible value after which the compiler won't be able to fit the program in the switch. For each scenario, we track the number of MAT entries and corresponding memory usage. We provide the memory usage only in terms of Static Random Access Memory (SRAM) since in our S4 implementation we do not require any Ternary Content Addressable Memory (TCAM). According to the results presented in Figure 7, we conclude that in general count-based windows offer better scalability than time-based windows. For both count-based and time-based windows, the tumbling model offers better scalability and less memory usage than the sliding windows model. In the best scenario which is count-based tumbling windows, S4 can accommodate up to $457k$ operator instances which is 3 orders of magnitude higher than the software-based solutions, while handling up to $286k$ concurrent streams. For time-based windows, we obtain up to $362k$ operator instances and $65k$ concurrent streams. For the highest parallelism degree, we estimate the maximum memory usage of our implementation to 270 MBytes of SRAM.

3) *Performance Evaluation:*

a) ***Throughput:*** In this experiment, we use T-Rex to send the traffic to the S4 switch on a 10 Gbps ingress port and we vary the incoming line rate from 10% to 100% of the link speed. Then, we measure the aggregated throughput of the outgoing traffic on the S4 egress ports. We perform the same measurement for each of the windowing models `cbtw1`, `cbsw1`, `tbtw1` and `tbsw1`, described in Table II and Table III. In this scenario, we use a packet size of 512 Bytes. The measured incoming rate is 9.55 Gbps when a 100% of the link speed is set in T-Rex. The obtained results are presented in Figure 8. First, we note that in the case of tumbling windows both count-based and time-based windows have a similar output throughput which is the same as the incoming throughput. However, in the case of sliding windows S4 acts as an amplifier, i.e., a throughput multiplier because of the overlapping windows and associated packet replication. For instance, with `cbsw1`, the generated throughput is 4 times higher than the incoming rate while with `tbsw1`, it is 2 times higher. This is explained by the overlap degree which influences how much the incoming stream gets amplified as in `cbsw1` with $\Sigma(n = 5, \delta = 1)$ resulting in a $maxOverlap = n - \delta = 4$ and in the case

Bochra Boughzala, and Boris Koldehofe. **In-Network Management of Parallel Data Streams over Programmable Data Planes**. In Proceedings of the 23rd International Federation for Information Processing (IFIP) Networking 2024 Conference (NETWORKING 2024), IFIP, 9 pages, 2024.

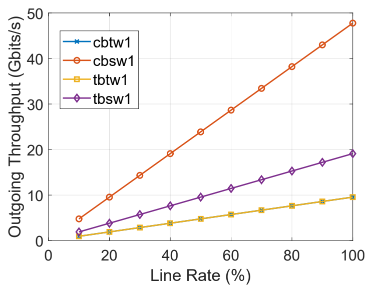


Figure 8: Throughput

Measurement	`cbtw`	`cbsw`	`tbtw`	`tbsw`
Avg Latency	1.76 μs	1.86 μs	1.75 μs	1.87 μs
Min Latency	1.72 μs	1.8 μs	1.72 μs	1.83 μs
Max Latency	1.8 μs	1.92 μs	1.8 μs	1.93 μs

Table V: Latency Overhead

of `tbsw1` with $\Sigma(\Delta = 1000, \tau = 500)$ resulting in a $maxOverlap = \Delta/\tau = 2$. With `cbsw1`, when the incoming rate is set at 100%, the total measured outgoing throughput is 47.76 Gbps which corresponds to 11 millions events per second for one particular stream.

*b) **Latency Overhead:*** The measurement of the latency overhead in S4 was conducted by using P4 Stamper (P4STA) [14], [15], a P4-based framework that provides high precision latency measurements in the data plane with nanoseconds accuracy. To measure the packet delay in S4 from ingress to egress, we deploy S4 as the Device Under Test (DUT) on a first Tofino switch and P4STA on a second Tofino switch. Both are directly connected, such as P4STA is placed between T-Rex, the traffic generator and our S4 switch. To compute the latency, P4STA augments the original traffic with local timestamps on the way in and out of S4. We perform this experiment for each of the windowing semantics. The obtained results from P4STA are presented in Table V. We observe that for all the window models the latency remains under $2\mu s$.

*c) **Control Plane Overhead:*** Lastly, we evaluate the control plane overhead by measuring the configuration time for different parallelism degrees for the tumbling and sliding window models. For each data point, we perform the measurement 100 times and compute the average value. The window type, i.e., count-based or time-based, does not have an effect on the configuration time as it is expressed in the same one MAT entry. Therefore, we focus on the difference between having overlapping or non-overlapping windows. The results presented in Figure 9 show that with higher parallelism degrees there is a higher control plane overhead. As the number of operator instances increases the configuration time also increases for both models but not in the same way. In the case of sliding windows the configuration time is higher than the tumbling windows because of the required packet replication and associated multicast groups that must be created and configured in addition to the MAT entries of the additional

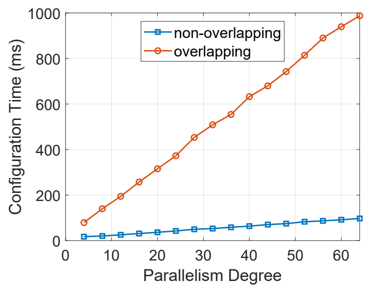


Figure 9: Control Plane Overhead

slidingWindows table.

VI. Related Work

Boughzala et al. highlight in [20] the potential benefits of in-network computing for data-analytics. In [21], they show initial findings for count-based windows excluding concrete resource requirements on real hardware. Contrary, in S4 we demonstrate that advanced windowing semantics are feasible on real hardware implementation and we carefully analyze corresponding resource consumption and performance characteristics. While S4's focus is the scalability in parallel stream processing, other approaches aim at improving a particular function in Distributed Event-Based System (DEBS) such as pattern matching in Complex Event Processing (CEP). For instance, P4CEP [25] is a framework for accelerating the execution of CEP queries over P4 programmable data planes. FastReact-PS [26] is another P4-based framework for efficient complex-event detection for Publish/Subscribe (PubSub) systems. Other works focus on reducing the Input/Output (I/O) overhead with other in-network computing models, e.g., DPDK and Remote Direct Memory Access (RDMA). For example, an RDMA-based implementation of Apache Storm was proposed in [28] showing how RDMA helps overcoming the performance limitations of Netty (the communication component of Apache Storm) by eliminating frequent memory copies and context switching which resulted in increased throughput and reduced CPU usage. Typhoon [27] is an SDN-based framework which relies on a DPDK version of Open vSwitch (OVS) for supporting real-time stream processing. It includes an SDN controller application to adapt to changes in the workload by modifying the routing policy and changing data analytics pipeline topology at run-time via the Openflow interface. TCAM-based filtering is another in-network acceleration method which supports line-rate event matching as described by Tariq et al. in [17] where the authors propose PLEROMA, an SDN-based PubSub middleware which utilizes TCAM matching to achieve bandwidth efficiency and low-latency event filtering.

Outside in-network computing, general-purpose optimizations for stream processing [29] are widely covered [16] [18] and can be used in conjunction with the proposed S4 framework.

VII. Conclusion and Future Work

In this paper, we presented S4 a Stateful and Scalable Splitter Switch supporting in-network window-based parallel stream processing. We proposed the data plane algorithms of four essential windowing semantics, i.e., count-based and time-based windows both in the tumbling and sliding mode. We developed a prototype of S4 using the $P4_{16}$ programming language and performed the evaluation of our implementation on real hardware, i.e., Intel Tofino switch. The resource consumption analysis shows that S4 handles up to $286k$ concurrent data streams with a parallelism degree of up to $457k$ operator instances, which is 3 orders of magnitude higher than state-of-the-art solutions.

As future work, we are interested in implementing S4 in a real-world scenario which would require addressing the potential integration challenges. Specifically, the data-analytics logic resides at the application layer while S4 is executed in the network layer. Moreover, S4 is built on the assumptions of one event per packet and in-order arrival of events while there could be out-of-order events and packets with multiple events. Therefore, it is essential to test S4 with real-world dataset to ensure its integration with stream processing applications does not introduce errors in the results.

Acknowledgment

This work has been supported by the German Research Foundation (DFG) as part of the project C2 within the Collaborative Research Center (CRC) 1053 MAKI. The authors want to thank Vladimir Gurevich for his feedback on an earlier version of this paper. We also thank Ralf Kundel for his help in using P4Sta.

References

[1] Röger, H. and Mayer, R., 2019. A comprehensive survey on parallelization and elasticity in stream processing. ACM Computing Surveys (CSUR), 52(2), pp.1-37.
[2] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K., 2015. Apache flink: Stream and batch processing in a single engine. The Bulletin of the Technical Committee on Data Engineering, 38(4).
[3] Amann, M., 2018. Efficient splitter for data parallel complex event processing (Bachelor's thesis).
[4] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. and Walker, D., 2014. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3), pp.87-95.
[5] Dang, H.T., Canini, M., Pedone, F. and Soulé, R., 2016. Paxos made switch-y. ACM SIGCOMM Computer Communication Review, 46(2), pp.18-24.
[6] Sapio, A., Canini, M., Ho, C.Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D. and Richtárik, P., 2021. Scaling distributed machine learning with In-Network aggregation. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21) (pp. 785-808).
[7] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C. and Stoica, I., 2017, October. Netcache: Balancing key-value stores with fast in-network caching. In Proceedings of the 26th Symposium on Operating Systems Principles (pp. 121-136).
[8] Intel®. 2024. Tofino1 6.5 Tbps. Retrieved January 2024 from https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html
[9] $P4_{16}$ Language Specification. Retrieved January 2024 from https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
[10] DEBS 2024. Grand Challenge. Retrieved April 2024 from https://2024.debs.org/call-for-grand-challenge-solutions/
[11] PyFlink. Windowing in Stream Processing. Retrieved April 2024 from https://ofili.medium.com/windowing-in-stream-processing-8bb636bdfaa7
[12] TRex. Realistic Traffic Generator. Retrieved January 2024 from https://trex-tgn.cisco.com/
[13] DPDK. Data Plane Development Kit. Retrieved January 2024 from https://www.dpdk.org/
[14] Kundel, R., Siegmund, F., Blendin, J., Rizk, A. and Koldehofe, B., 2020, April. P4STA: High performance packet timestamping with programmable packet processors. In NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium (pp. 1-9). IEEE.
[15] Kundel, R., Siegmund, F., Hark, R., Rizk, A. and Koldehofe, B., 2022. Network testing utilizing programmable network hardware. IEEE Communications Magazine, 60(2), pp.12-17.
[16] Mayer, R., Tariq, M.A. and Rothermel, K., 2017, June. Minimizing communication overhead in window-based parallel complex event processing. In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (pp. 54-65).
[17] Tariq, M.A., Koldehofe, B., Bhowmik, S. and Rothermel, K., 2014, December. PLEROMA: A SDN-based high performance publish/subscribe middleware. In Proceedings of the 15th International Middleware Conference (pp. 217-228).
[18] Mayer, R., Koldehofe, B. and Rothermel, K., 2015. Predictable low-latency event detection with parallel complex event processing. IEEE Internet of Things Journal, 2(4), pp.274-286.
[19] De Matteis, T. and Mencagli, G., 2017. Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. International Journal of Parallel Programming, 45(2), pp.382-401.
[20] Boughzala, B. and Koldehofe, B., 2021, June. Accelerating the performance of data analytics using network-centric processing. In Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (pp. 192-195).
[21] Boughzala, B., Gärtner, C. and Koldehofe, B., 2022, July. Window-based Parallel Operator Execution with In-Network Computing. In The 16th ACM International Conference on Distributed and Event-based Systems (DEBS'22) (pp. 91-96). ACM New York, NY, USA.
[22] Gebara, N., Lerner, A., Yang, M., Yu, M., Costa, P. and Ghobadi, M., 2020, November. Challenging the stateless quo of programmable switches. In Proceedings of the 19th ACM Workshop on Hot Topics in Networks (pp. 153-159).
[23] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M. and Kalnis, P., 2017, November. In-network computation is a dumb idea whose time has come. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks (pp. 150-156).
[24] Kim, D., Jain, A., Liu, Z., Amvrosiadis, G., Hazen, D., Settlemyer, B. and Sekar, V., 2020. Unleashing in-network computing on scientific workloads. arXiv preprint arXiv:2009.02457.
[25] Kohler, T., Mayer, R., Dürr, F., Maaß, M., Bhowmik, S. and Rothermel, K., 2018, August. P4CEP: Towards in-network complex event processing. In Proceedings of the 2018 Morning Workshop on In-Network Computing (pp. 33-38).
[26] Vestin, J., Kassler, A., Laki, S. and Pongrácz, G., 2020. Toward in-network event detection and filtering for publish/subscribe communication using programmable data planes. IEEE Transactions on Network and Service Management, 18(1), pp.415-428.
[27] Cho, J., Chang, H., Mukherjee, S., Lakshman, T.V. and Van der Merwe, J., 2017, November. Typhoon: An SDN enhanced real-time big data streaming framework. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (pp. 310-322).
[28] Zhang, Z., Liu, Z., Jiang, Q., Chen, J. and An, H., 2021. RDMA-based apache storm for high-performance stream data processing. International Journal of Parallel Programming, 49(5), pp.671-684.
[29] Hirzel, M., Soulé, R., Schneider, S., Gedik, B. and Grimm, R., 2014. A catalog of stream processing optimizations. ACM Computing Surveys (CSUR), 46(4), pp.1-34.