Unraveling the Impact of Window Semantics: Optimizing Join Order for Efficient Stream Processing

Ariane Ziehn
TU Berlin, BIFOLD, Germany
ariane.ziehn@tu-berlin.de

Jan Szlang*
Snowflake Computing, Germany
jan.szlang@snowflake.com

Steffen Zeuch
TU Berlin, BIFOLD, Germany
steffen.zeuch@tu-berlin.de

Volker Markl
TU Berlin, BIFOLD, Germany
volker.markl@tu-berlin.de

ABSTRACT

Window joins (WJs) are fundamental operators in stream processing systems (SPSs), enabling continuous, time-aware joins over unbounded data streams. Unlike time-agnostic relational joins, WJs incorporate temporal semantics associated with different window types (i.e., sliding, session, and interval windows), which introduce uncertainty in algebraic properties such as commutativity and associativity. As a result, state-of-the-art SPSs exploit only a single, fixed join order, which limits optimization opportunities and often leads to suboptimal performance. In this work, we eliminate this restriction by introducing three transformation rules that enable WJ reordering while preserving query semantics for those window types. Based on them, we propose *WJR*, an algorithm that systematically enumerates semantically equivalent join orders, expanding the search space for finding efficient WJ execution plans. Our evaluation shows speedups of up to 10 for multi-way WJ queries under various window configurations and rate ratios, highlighting the performance benefits of flexible join reordering in streaming queries.

PVLDB Reference Format:
Ariane Ziehn, Jan Szlang, Steffen Zeuch, and Volker Markl. Unraveling the Impact of Window Semantics: Optimizing Join Order for Efficient Stream Processing. PVLDB, 18(8): 2468 - 2481, 2025.
doi:10.14778/3742728.3742741

PVLDB Artifact Availability:
The source code, data, and artifacts have been made available at (1) github.com/arianeziehn/reorder_window_joins (Case Study and Evaluation in Flink) and (2) github.com/szlangini/WindowJoinReordering (*WJR*).

1 INTRODUCTION

Stream processing systems (SPSs) enable continuous data analytics over unbounded data streams by maintaining standing queries. As the Internet of Things (IoT) expands, both the volume of data and the number of streams processed by SPSs grow exponentially, leading to increasingly complex analytical queries [20, 67, 68]. A major source of complexity is the need to correlate multiple streams through

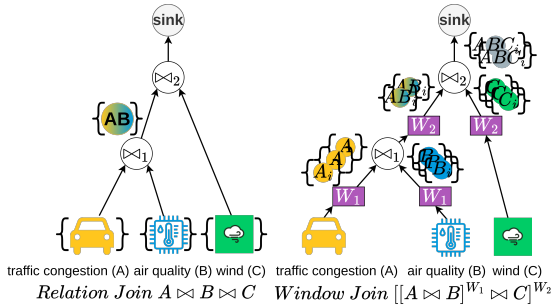


Figure 1: Multi-Way Relational Join versus Multi-Way WJ.

multi-way window joins (WJs), which are essential for applications such as moving object tracking [29, 35], online auctions [58, 62], smart manufacturing [53, 59], and property monitoring [60]. To manage unbounded streams, SPSs utilize WJs, which perform time-based joins over finite intervals defined by various window semantics, e.g., overlapping sliding windows or non-overlapping tumbling windows [14, 19, 37, 63]. For example, consider a city-wide traffic management application [29, 35, 53] depicted in Figure 1, which processes three streams: high-frequency traffic congestion data (*A*), moderate-frequency air-quality metrics (*B*), e.g., temperature or particulate matter, and low-frequency wind data (*C*).

Example 1. *In this scenario, the system dynamically adjusts speed limits based on congestion and environmental conditions, e.g., reducing speeds when temperatures drop below zero or wind speeds are high. To this end, a query with an hourly sliding window starting every five minutes joins these streams to derive speed recommendations. Joining the high-frequency congestion data with the moderate-frequency air quality data first yields a high-frequency intermediate result, leading to computational overhead for the subsequent WJ with wind data. A better plan would produce a smaller intermediate result by first joining the high-frequency congestion data with the sparser wind data, reducing processing overhead for the subsequent WJ.*

This scenario highlights the critical impact of WJ ordering for performance in SPSs [14, 19, 37, 43]. This is in line with findings in relational databases, where multi-way joins are known to be computationally intensive and order-sensitive [41, 47]. In databases, a query optimizer typically explores a large *search space* of equivalent join plans to find an efficient execution strategy. These equivalences arise from algebraic properties such as *commutativity* and *associativity*, which make it possible to transform one query plan into another without altering query semantics [11, 33]. By enumerating

*Work performed while at TU Berlin.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742741

different plans and applying a *cost function* to estimate their execution efficiency, relational optimizers prune the search space and select the best plan based on its costs [17, 42, 54]. As a result, join reordering is a cornerstone for achieving high performance.

Although the underlying principles of relational optimizers carry over conceptually to streaming queries, WJs introduce additional complexity due to their time-based semantics. In particular, unlike time-agnostic relational joins over finite data, WJs continuously process infinite data as part of a standing query, necessitating time-based semantics. Figure 1 illustrates this contrast: The relational join $\bowtie_1$ first processes the sets A and B, producing an intermediate result AB, which is then joined with C in a subsequent step before the query completes. In contrast, each WJ pairs a join and a window operator, e.g., $\bowtie_1$ with W_1. W_1 is responsible for discretizing the streams A and B into a sequence of finite substreams (sets) A_i and B_i, which then $\bowtie_1$ continuously composes. Therefore, windowing is not just an auxiliary function for discretizing streams but directly determines which tuples are eligible for each join operation, potentially invalidating algebraic properties. For instance, reordering two WJs, as proposed in Example 1, may change which tuples fall in overlapping time intervals and thus lead to missing or incorrect results. As a consequence, state-of-the-art SPSs lack the capability to systematically reorder WJs, as the conditions under which commutativity and associativity hold remain unclear across different window types. In other words, the *search space* of execution plans for multi-way WJs is practically limited to a single plan derived from the input query. As a result, the SPS optimizer has no freedom to pick a better plan even if it exists, thereby leaving significant opportunities for performance gains on the table [5, 19, 31, 58].

In this paper, we address the fundamental question of whether and how WJs can be reordered without changing query semantics, thereby unlocking a larger search space of execution plans for stream optimization. To this end, we first analyze algebraic properties of WJs across commonly used window types (i.e., sliding, session, and interval windows) and time domains (i.e., processing and event time). Second, we devise generic *transformation rules* to enable these reorderings under well-defined conditions. Third, we propose an enumeration algorithm, *WJR*, that systematically constructs valid join orders for a given multi-way WJ query. In sum, by focusing on *search-space expansion*, we lay the groundwork for future WJ optimization and make the following contributions:

- We provide a *comprehensive commutativity and associativity analysis* of common WJ types in SPSs, revealing which reorderings preserve semantics (Sections 3 and 4).
- We introduce *three transformation rules* that enable WJ reordering, even in cases where commutativity and associativity do not inherently hold (Section 5).
- We propose the *WJR* algorithm, which systematically enumerates semantically valid multi-way WJ plans, thereby lifting the restriction of a single join order and expanding the search space for future WJ optimization (Section 6).
- We *evaluate* our approach under diverse workloads and window semantics, demonstrating that reordering yields significant throughput and resource benefits (Section 7).

We discuss related work in Section 8 and conclude in Section 9.

2 PRELIMINARIES

In this section, we provide an overview of stream processing fundamentals in Section 2.1 and detailed concepts for WJs in Section 2.2.

2.1 Stream Processing Fundamentals

Streams and Tuples. A stream S is a continuous list of tuples generated by data sources such as sensors [2, 3, 27]. A tuple t is a list of attributes $\mathrm{t}(a_1, ...a_n)$, including a timestamp ts, which holds the relevant time information for windowing [3, 27, 38]. In particular, SPSs have two different notions of time: processing and event time [2, 61]. Event time uses the creation time of the tuple outside the SPS as the timestamp ts of a tuple t, whereas processing time uses the system clock when the tuple is processed in the SPS. We write $t.ts$ for the timestamp of t and consider it an accessible tuple value regardless of the time domain. All tuples $t_i \in S$ share the same list of attributes, i.e., a common schema $S(a_1, ..., a_n)$.

Queries and Operators. Users specify operations on streams in queries. In SPSs, a query is represented as a directed acyclic graph (DAG) comprising operators as nodes. Formally, a query is defined as a tuple $Q = (OP, \lambda, W, \omega)$ over the set of operators OP, the set of window operators W, the partial function $\lambda : OP \nrightarrow OP^n, n \in [1, 2]$ which assigns child nodes to an operator, and the function $\omega : OP^2 \rightarrow W$ which assigns a window operator W_b to a binary operator $OP^2_b, b \in \mathbb{N}$ [4, 50]. A query plan ρ derives the output tuples of Q by executing its operators along the DAG to the sink [21, 39]. Streams act as sources, and each operator OP_u sends its output to its parent OP_p, where the associated W_p discretizes the output of OP_u if $OP_p \in OP^2$. WJs are such binary operators OP^2 paired with a window operator W, written as $[A \bowtie_\theta B]^W = AB$, where A and B are input streams and AB is the output stream of joined tuple pairs (a, b). Here, W discretizes each infinite input stream S into a sequence of finite substreams S_i based on its semantics [2, 6]. A query Q can be represented by multiple semantically equivalent query plans $\rho_k, k \in \mathbb{N}^+$, each producing the same output despite variations in execution order or structure. We adopt the definition of Negri et al. [45], where two plans are semantically equivalent if, for all input tuples, they produce the same output after execution and duplicate elimination. The cost of a plan c_{ρ_k} determines its quality, e.g., in terms of resource usage or runtime.

2.2 Window Join Types and their Semantics

SPSs support different types of windows, each defining specific semantics for WJs according to the selected window type. Table 1 summarizes the available WJ types in common SPSs, i.e., tumbling, sliding, session, and interval. Time-based windows, i.e., tumbling and sliding windows, are foundational in SPSs for managing the temporal aspects of continuous data streams and are thus widely adopted in SPSs and application scenarios [2, 6, 14, 53, 64]. Early SPSs primarily operated on processing time (PT), handling tuples in the order of their arrival. However, as data sources increasingly originated outside the cloud, the necessity to tackle transmission latency led to the widespread adoption of event time (ET) [3]. Since then, event time has driven the development of more advanced window types, such as content-based (or data-driven) session and interval windows, to meet diverse application requirements. Today, event time is the dominant paradigm for cloud-based SPSs [2, 7, 24].

Table 1: Overview of WJ Types in common SPS.

WJ Type	Parameters	Categories			SPSs					
		Creation	Measure	Time Domain	Apache Flink [23]	NebulaStream [68]	Apache Spark [66]	Apache Beam [22]	Apache Storm [25]	Azure SA [1]
Tumbling Window Join	length l	time	time	PT	✔	✗	✗	✔	✔	✔
				ET	✔	✔	✔	✔	✔	✔
Sliding Window Join SWJ	length l, slide s	time	time	PT	✔	✗	✗	✔	✔	✔
				ET	✔	✔	✔	✔	✔	✔
Session Window Join SessWJ	gap g	content	time	PT	✔	✗	✗	✔	✗	✔
				ET	✔	✗	✗	✔	✗	✔
Interval Join IVJ	lower bound lB upper bound uB	content	time	ET	✔	✗	(✔)	✗	✗	✗

Legend: () = limited options, PT = processing time, ET = event time, SA Stream Analytics

Window Join Semantics. Each window operator has two semantic aspects: (1) intra-window and (2) inter-window semantics.

(1) Intra-window semantics define which tuples are assigned to each finite substream(s) S_i. For all considered WJ types in Table 1, window parameters are measured in time. Thus, each S_i corresponds to a time interval (i.e., a window) $w_i = [ts_{b_i}, ts_{e_i})$, where ts_{b_i} denotes the beginning and ts_{e_i} the end of w_i [1, 2, 28, 63]. A tuple t is assigned to S_i if $t.ts \in w_i$. Furthermore, intra-window semantics incorporate the join operation, where each tuple $a \in A_i$ is paired with a tuple $b \in B_i$ to produce the pair (a, b) if they satisfy the join predicate θ [13, 19, 34, 57]. Formally,

$$[A \bowtie_\theta B]^{w_i} = A_i \bowtie_\theta B_i = \{(a,b) | a \in A_i \wedge b \in B_i \wedge \theta_{AB}\}$$

For simplicity, we assume that a join predicate θ exists and join operations are independent, omitting further notation.

(2) Inter-window semantics define how subsequent windows are created, i.e., how the stream is partitioned into substreams [64]. When multiple window operators W_n occur in a query Q, we denote the corresponding window as w_i^n.

Window Join Types. We investigate in detail the semantic differences among these WJ types:

(1) Tumbling and Sliding Window Joins (SWJs) compute join operations over windows w_i with a fixed length $l = ts_{e_i} - ts_{b_i}$ that are aligned with the logical clock, thereby ensuring consistent and predictable join operations [2, 14, 28, 61]. For instance, a tumbling WJ with a 1-hour length submitted at 6:25 would align its first window to the clock w_1 = [6:25,7:00), as shown in Figure 2. Subsequent windows are derived based on type-specific inter-window semantics. In event time, logical clock alignment is based on the timestamp of the first tuple that occurs. As a result, the same window operation is applied uniformly to both streams, leading to identical windows w_i for both substreams A_i and B_i over all partitions [57].

Tumbling windows divide streams into non-overlapping substreams, where each window w_i starts immediately after the end of its predecessor w_{i-1}, i.e.,$ts_{e_{i-1}} = ts_{b_i}$. For instance, our example in Figure 2 could be used for periodic correlations such as hourly sales or energy consumption reporting [53] with w_2 = [7:00,8:00). In contrast, *sliding windows* (or hopping windows [1, 52]) employ an additional fixed parameter, the so-called slide s. Thus, SWJs create a sequence of potentially overlapping substreams $S_{i+n} = [S]^{ts_{e_{i+n}}}_{ts_{b_{i+n}}}$, where $ts_{i+n} = ts_i + n \cdot s$, respectively $(i, n, s \in \mathbb{N})$ [14, 28]. This overlap enables a more fine-grained analysis of fast-changing data, for example, by analyzing a 5-minute window that slides every minute, as required in applications such as traffic management [35] or fraud detection [52]. The overlapping nature of sliding windows can generate duplicate output tuples and increase resource usage compared to their non-overlapping counterparts [25, 37, 51]. Since tumbling windows are a special case of sliding windows with $s = l$, we focus our analysis on SWJs, formally denoted as $SW(l, s)$.

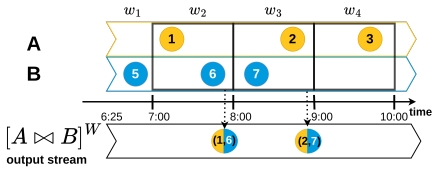


Figure 2: Logical Clock Aligned Tumbling WJ in PT.

(2) Session Window Joins (SessWJs) group tuples into non-overlapping windows (sessions) based on a gap g, which defines the minimum period of inactivity required to separate sessions [22, 64]. In particular, a session w_i ends at $t_{N-1}.ts$, and the next window w_{i+1} starts at $t_N.ts$ if $t_{N-1}.ts - t_N.ts \geq g$, where t_N is the last seen tuple and session boundaries are inclusive. We note that the actual gap between sessions can exceed g and further denote a session window as $SessW(g)$ [61, 64]. Like SWJs, SessWJs apply the same session w_i to both substreams A_i and B_i. However, SessWJs are not logical clock aligned, and key-based partitions may have different session boundaries [2, 22, 23]. For gap computation, SessWJs require tuples across the stream partitions of A and B to be temporally ordered, i.e., $t_n \in A \cup B : t_{n-1}.ts \geq t_n.ts$. Its properties make SessWJs particularly suited for handling irregular data arrivals, such as dynamic user session analytics in domains like e-commerce and media streaming [22, 52].

(3) Interval Joins (IVJs) pair an incoming tuple from the left stream with all temporally relevant tuples from the right stream using content-based windows w_i defined by a lower bound (lB) and an upper bound (uB). For example, in the WJ $[A \bowtie B]^W$, each tuple $a_i \in A$ is associated with a window $w_{i_B} = [a_i.ts - lB, a_i.ts + uB]$ applied to stream B. We denote an interval as $I(lB, uB)$. Although less common, this mechanism is particularly valuable in applications that require precise temporal alignment and the avoidance of costly overlaps, such as algorithmic trading [56] or aligning IoT sensors [16, 53]. The boundaries of w_{i_B} can be in- or exclusive [23]. For simplicity, we consider inclusive boundaries for further analysis while noting that our approach generalizes to exclusive boundaries.

Time Propagation in Multi-Way Window Joins. Stateful window operations create new tuples that require a timestamp if they contribute to a subsequent windowed operation. We consider the time propagation as an additional window parameter, like length or slide. Different strategies exist for propagating time between window operators: In processing time, all output tuples t_j of one window w_i are assigned to its largest timestamp [1, 6]. Formally, $\forall t_j \in w_i : t_j.ts = ts_{e_i} - \delta$, where $\delta = 1$ for exclusive, and $\delta = 0$ for inclusive bounds. This strategy follows the processing time assumption that a tuple is assigned to the time it first occurs in the SPS. The tumbling WJ in Figure 2 is executed in processing time.

In event time, without user interaction, all output tuples are assigned to the current system time, i.e., processing time [23]. To prevent changing the time domain, the user can reassign the timestamp of the output stream after each WJ, i.e., either $a.ts$ or $b.ts$ as the timestamp for tuple t_j of AB. Formally, $t_j.ts = a.ts \vee b.ts$.

3 COMMUTATIVITY

In this section, we examine the commutativity of WJ types, a key requirement for join reordering, as it ensures that altering the join order does not affect query results. We formulate Hypothesis 1, drawing on relational join principles, and analyze it for SWJs in Section 3.1, SessWJs in Section 3.2, and IVJs in Section 3.3.

Hypothesis 1. *A WJ operator is commutative if, for every output tuple (a,b) detected by the query $Q_1 = [A \bowtie B]^W$, a corresponding output tuple is detected by $Q_2 = [B \bowtie A]^W$, and vice versa.*

3.1 Sliding Window Joins

Case C1. For time-based SWJs, the tuple (a, b) is a valid output of Q_1 if $a.ts, b.ts \in w_i = [ts_{b_i}, ts_{e_i})$, and a valid output of Q_2 if $a.ts, b.ts \in w_j = [ts_{b_j}, ts_{e_j})$.

Proof. The time intervals of w_i and w_j are defined by the parameters of the window operator W and aligned to the logical clock. Thus, w_i and w_j are created irrespective of the tuple content in the streams A or B. It follows that for the same W, the sequence of windows in both queries is equivalent, i.e., $W \in q_1 = W \in q_2 \rightarrow w_i = w_j$. Thus, every tuple $a \in A$ and $b \in B$ is assigned to the same time interval, and if $a.ts, b.ts \in w_i \rightarrow a.ts, b.ts \in w_j$. For this reason, the join order is irrelevant, and every output tuple (a, b) of Q_1 is also an output tuple of Q_2. □

Theorem 1. *SWJs are commutative.*

3.2 Session Window Joins

Case C2. For content-based SessWJs, the tuple (a, b) is a valid output of Q_1 if $a.ts, b.ts \in w_i = [ts_{b_i}, ts_{e_i}]$, and a valid output of Q_2 if $a.ts, b.ts \in w_j = [ts_{b_j}, ts_{e_j}]$.

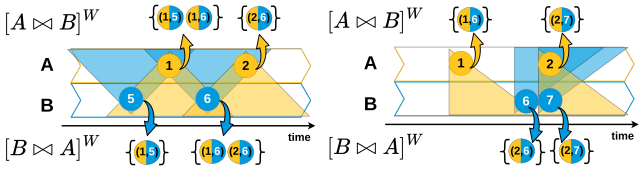
Proof. The time intervals of w_i and w_j are defined by the temporal order of tuples $t_n \in A \cup B$ and the gap between each tuple pair (t_{n-1}, t_n). It follows that $w_i = w_j$ as the join order does not affect $A \cup B$. Thus, every tuple $a \in A$ and $b \in B$ is assigned to the same time interval, and if $a.ts, b.ts \in w_i \rightarrow a.ts, b.ts \in w_j$. For this reason, the join order is irrelevant, and every output tuple (a, b) of Q_1 is also an output tuple of Q_2. □

Theorem 2. *SessWJs are commutative.*

3.3 Interval Joins

Content-based IVJs create windows based on tuples from the left join side and the window bounds, lB and uB. Thus, the tuple (a, b) is a valid output of Q_1 if $b.ts \in w_i = [a.ts - lB, a.ts + uB]$, and a valid output of Q_2 if $a.ts \in w_j = [b.ts - lB, b.ts + uB]$. We analyze two cases, Case $C3$ with equal-sized bounds ($lB = uB$) and Case $C4$ with unequal-sized bounds ($lB \neq uB$), depicted in Figure 3. In our proof, we investigate Q_2, where tuples of B define w_j and examine edge cases to verify if (a, b) detected in Q_1 is also detected in Q_2:
Case C3. We analyze IVJs with equal-sized bounds ($lB = uB$).

Proof. (1) $b.ts = a.ts - lB \rightarrow w_j = [a.ts - 2 \cdot lB, a.ts - lB + uB]$. Thus, (a, b) is detected in Q_2 as $a.ts = a.ts - lB + uB \rightarrow a.ts \in w_j$.
(2) $b.ts = a.ts + uB \rightarrow w_j = [a.ts + uB - lB, a.ts + 2 \cdot uB]$. Thus, (a, b) is detected in Q_2 as $a.ts = a.ts + uB - lB \rightarrow a.ts \in w_j$. □



(a) IVJ Case C3 with lB = uB **(b) IVJ Case C4 with lB < uB**

Figure 3: Commutativity Analysis of IVJs.

Theorem 3. *IVJs with equal-sized bounds are commutative.*

Case C4. We analyze IVJs with unequal-sized bounds ($lB \neq uB$). We examine $lB < uB$, though our proof also holds for $lB > uB$.

Proof. (1) $b.ts = a.ts - lB \rightarrow w_j = [a.ts - 2 \cdot lB, a.ts + uB - lB]$. Thus, the tuple (a, b) is detected in Q_2 because $lB < uB \rightarrow a.ts \in w_j$, i.e., $a.ts < a.ts + uB - lB$.
(2) $b.ts = a.ts + uB \rightarrow w_j = [a.ts + uB - lB, a.ts + 2 \cdot uB]$. Thus, the tuple (a, b) is not detected in Q_2 because $lB < uB \rightarrow a.ts \notin w_j$, i.e., $a.ts < a.ts + uB - lB$. □

Theorem 4. *IVJs with unequal-sized bounds are not commutative.*

Figure 3 shows both cases with the output of Q_1 (yellow intervals) on top and Q_2 (blue intervals) at the bottom. Figure 3a supports the commutativity of Case C3 (equal-sized bounds), as both queries yield identical outputs. In contrast, Figure 3b presents a counterexample for Case C4, where differences in the outputs of Q_1 and Q_2 reveal that assuming commutativity leads to incorrect reorderings that alter query semantics, resulting in missing or additional matches. To showcase this, consider the following adaptive gate management scenario [15]:

Example 2. *An airline uses flight arrival data (A) combined with gate availability (B) to decide whether to extend gate openings for delayed flights. While Q_1 correctly identifies two cases where the plane could wait, the reordered Q_2 misses a critical match (tuple (a_1, b_6)) and erroneously produces an extra result (tuple (a_2, b_6)).*

4 ASSOCIATIVITY

We extend our WJ analysis to associativity, another key property for join reordering. Associativity ensures that the grouping of join operations can be modified without altering query results. To this end, we examine a three-way WJ, where the first WJ, i.e., $[A \bowtie B]^{W_1} = AB$, was analyzed in Section 3. The second WJ composes a tuple $(a, b) \in AB$ with a tuple $c \in C$ to the result tuple (a, b, c) if both fulfill the WJ constraints. We formulate Hypothesis 2, drawing on relational join principles, and analyze it for SWJs in Section 4.1, SessWJs in Section 4.2, and IVJs in Section 4.3.

Hypothesis 2. *A WJ operator is associative if, for each output tuple (a, b, c) detected by $Q_1 = [[A \bowtie B]^{W_1} \bowtie C]^{W_2}$, a corresponding output tuple is detected by $Q_2 = [A \bowtie [B \bowtie C]^{W_2}]^{W_1}$, and vice versa.*

4.1 Sliding Window Joins

The tuple (a, b, c) is valid output of Q_1 if $a.ts, b.ts \in w_i^1 = [ts_{b_i}, ts_{e_i})$, and $ab.ts, c.ts \in w_j^2 = [ts_{b_j}, ts_{e_j})$. In the remainder, we analyze four cases defined by the combinations of window length l and slide s.
Case A1. We consider two equivalent overlapping sliding window operators, i.e., $W_1 = W_2$ and $s < l$.

Processing Time. The time propagation between W_1 and W_2 is defined as follows: $\forall(a,b) : (a,b).ts = ts_{e_i} - 1$. Furthermore, (a,b) and c must occur in the same window w_j^2 to create (a,b,c) in Q_1. Following the inter-window semantics of SWJs, there exist $m = \frac{l}{s}$ windows w^2 that overlap with w_i^1. We visualize Case $A1$ in Figure 4 and depict the two overlapping windows w_{j-1} and w_{j+1} on top. We investigate two options for Q_2 to detect (a,b,c):

Proof. **Option (1):** $c \in C_j \rightarrow c.ts \in w_j^2$. Due to logical clock alignment, $w_j^2 = w_i^1$ and all timestamps $a.ts, b.ts, c.ts \in [ts_{b_i}, ts_{e_i})$. Thus, the tuple (a,b,c) is detected, regardless of the join order.

Option (2): $c \in C_{j+(m-1)} \rightarrow c.ts \in w_{j+(m-1)}^2 = [ts_{b_j} + n \cdot s \cdot (m-1), ts_{e_j} + n \cdot s \cdot (m-1)), (m > 1)$. Thus, the tuple c falls into one (or more) subsequent overlapping windows $w_{j+(m-1)}^2$ of w_j^2. These windows overlap by the time interval $w_{ol}^+ = [ts_{b_j} + s \cdot n \cdot (m-1), ts_{e_j})$. We highlight w_{ol}^+ in purple for the window w_j in Figure 4. If $c.ts \in w_{j+(m-1)}^2$, Q_2 detects an output tuple (b,c) in $[B \bowtie C]^{W_2}$ only if $b.ts \in w_{j+(m-1)}^2$. If $c.ts > ts_{e_j}$, it follows that c is assigned to the subsequent window $w_{j+(m-1)}^2$, but not to w_j^2. Formally, $c \in w_{j+(m-1)}^2$ and $c \notin w_j^2$, because $w_{ol}^+ \subset w_j^2, w_{j+(m-1)}^2$. If $b.ts < ts_{b_{j+(m-1)}}$, and thus, $b \in w_j^2$ and $b \notin w_{j+(m-1)}^2$, the tuple (a,b,c) is only detected in Q_1. □

Figure 4 provides a counterexample that refutes Hypothesis 2 for Case A1, demonstrating that the outputs of Q_1 and Q_2 are not equivalent. To showcase the consequences of such an invalid rewriting, consider the following window farm management scenario [53]:

Example 3. *In this application, wind turbine activity (C) is dynamically adjusted based on wind conditions (A) and nearby wildlife (B). The data is processed in 10-minute windows starting every 5 minutes. While Q_1 correctly aligns wind conditions and the presence of wildlife before adjusting turbine operations, the reordered Q_2 misses critical context (tuple (a_2, b_6, c_9)), potentially leading to unnecessary shutdowns or delayed wildlife protection.*

Event Time. The user propagates either $a.ts$ or $b.ts$ as the timestamp of AB. This extends the analysis of Case A1 in event time by Option (3) $c \in C_{j-(m-1)}$, alongside Options (1) and (2). We examine all three for each time propagation strategy:

Proof. **Propagation of A.ts.** If $a.ts$ is propagated as the timestamp of AB, only $a.ts$ and $c.ts$ are considered to assign tuples of AB and C to w_j^2 for $[AB \bowtie C]^{W_2}$ in Q_1.

Option (1): $c \in C_j \rightarrow c.ts \in w_j^2 = w_i^1$. Thus, all tuples fall into the same time interval, and (a,b,c) is detected in both queries.

Option (2): $c \in C_{j+(m-1)} \rightarrow c.ts \in w_{j+(m-1)}^2, (m > 1)$. Equivalent to Option (2) in processing time, there exists an overlap between each $C_{j+(m-1)}$ and C_j of the time interval $w_{ol}^+ = [ts_{b_j} + n \cdot s \cdot (m-1), ts_{e_j})$ (see Figure 4). We refer to Option (2) in processing time for a detailed analysis and conclude that an output tuple (a,b,c) detected by Q_1 is not necessarily detected by Q_2.

Option (3): $c \in C_{j-(m-1)} \rightarrow c.ts \in w_{j-(m-1)}, (m > 1)$. Option (3) investigates the overlap between ancestor substreams $C_{j-(m-1)}$ and C_j. This overlap is defined as follows: $w_{ol}^- = [ts_{b_j}, ts_{e_j} - n \cdot s \cdot (m-1))$. We apply the analysis of Option (2) respectively. In particular, if $c.ts < ts_{b_j}$ and $b.ts > ts_{e_{j-(m-1)}}$, it follows that $c \in w_{j-(m-1)}^2$ and $c \notin w_j^2$, while $b \notin w_{j-(m-1)}^2$ and $b \in w_j^2$. Thus, Q_2 does not detect (b,c) and misses the output tuple (a,b,c).

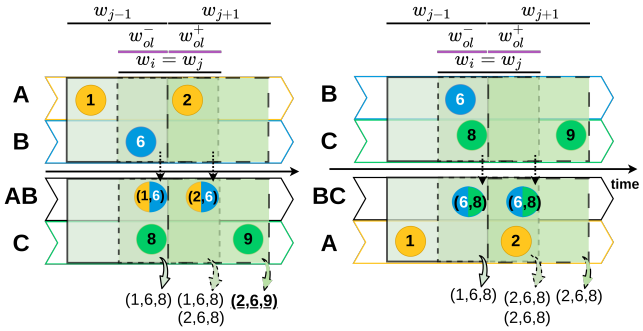


Figure 4: Case A1 of the SWJ with Q_1 (left) and Q_2 (right) in processing time and with slide $s = 0.5 \cdot l$.

Propagation of B.ts. In Q_1, tuples $a \in A$ and $b \in B$ first join under W_1, producing the intermediate results $(a,b) \in AB$. These tuples (a,b) then join with $c \in C$ under W_2, using only $b.ts$ for alignment. The output tuples $(a,b,c) \in ABC$ satisfy both join and window constraints within the respective windows w_i^1 and w_j^2. Since SWJs are logical clock aligned, the windows w_i^1 and w_j^2 are equivalent in Q_1 and Q_2, i.e., $W_1 = W_2 \rightarrow w_i^1 = w_j^2$. Thus, BC joins with A in Q_2 under the same window semantics as A joins B in Q_1, aligning $a \in A$ and $b \in B$ under W_1. By leveraging commutativity, as established in Theorem 1, each tuple (a,b,c) appears in both queries Q_1 and Q_2, independent of Options (1)-(3). □

We refer to the varying results in event time as *conditional associativity*, determined by the user-defined time propagation strategy.

Theorem 5. *SWJs of Case A1 are not associative in processing time and conditional associative in event time.*

Case A2. We consider two equivalent non-overlapping sliding window operators, i.e., $W_1 = W_2 \wedge s \geq l$.

Proof (sketch). Non-overlapping windows assign each tuple only to a single window. Thus, in contrast to Case $A1$ and regardless of the time domain, Case $A2$ solely contains Option (1) $c \in C_j$, where $a.ts, b.ts, c.ts \in w_i^1 = w_j^2$. As shown in Option (1) of Case $A1$, all output tuples in Q_1 are detected in Q_2.

Theorem 6. *SWJs of Case A2 are associative.*

Case A3. We consider two overlapping sliding window operators W_1 and W_2 with different lengths, i.e., $W_1.l \neq W_2.l \wedge s < l$.

Proof (sketch). Case $A1$ and $A3$ differ because the window lengths $W_1.l$ and $W_2.l$ are not equivalent. However, this does not affect the underlying proof structure of Case A1. Due to logical clock alignment, the same three options arise: Option (1) $c \in C_j \rightarrow c.ts \in w_j^2$, Option (2) $c \in C_{j+(m-1)}$, where $m = \frac{W_2.l}{s}$, and additional Option (3) $c \in C_{j-(m-1)}$ in event time. For processing time and event time with time propagation of $a.ts$, only Option (1) yields equivalent outputs, requiring each window of the window operator with the smaller length to be a subset of the larger one. If $b.ts$ is propagated in event time, Q_1 and Q_2 produce identical outputs.

We refer to the detailed proofs of Case A1 for a breakdown of the individual subcases.

Theorem 7. *SWJs of Case A3 are not associative in processing time and conditional associativity in event time.*

Case A4. We consider two non-overlapping sliding window operators with different lengths, i.e., $W_1.l \neq W_2.l \wedge s \geq l$. As windows are non-overlapping, only Option (1) $c \in C_j$ is of concern. We examine $W_1.l < W_2.l$, although our proof applies for $W_1.l > W_2.l$.

Processing Time. The time propagation between W_2 and W_1 in Q_2 is defined as follows: $\forall(b,c) : (b,c).ts = ts_{e_j} - 1$:

Proof. Let us consider a window $w_j^2 = [ts_{b_j}, ts_{e_j})$ aligned with the logical clock. Then, it exists an $w_i^1 = [ts_{b_j}, ts_{e_i})$ with $ts_{e_i} < ts_{e_j}$ because $W_1.l < W_2.l$. It follows that in Q_1, all pairs (a,b) of w_i^1 are assigned to w_j^2, while in Q_2, no pair (b,c) is assigned to w_i^1, because $ts_{e_j} - 1 > ts_{e_i} - 1$. As a result, the tuple (a,b,c) detected in Q_1 is not detected in Q_2. □

Event Time. The user propagates either $a.ts$ or $b.ts$ as the timestamp of AB. We investigate both time propagation strategies:

Proof. **Propagation of A.ts.** Due to logical clock alignment, the first join $[B \bowtie C]^{W_2}$ in Q_2 creates windows w_j^2, joining all tuples b and c within w_j^2, where $w_i^1 \subset w_j^2$. As a result, the constraints on $b.ts$ are relaxed, making the tuples (b,c) in Q_1 a subset of those in Q_2. Regardless of whether $b.ts$ or $c.ts$ is propagated, applying the smaller window operator afterward does not filter out these additional tuples, leading to additional output tuples (a,b,c) in Q_2.
Propagation of B.ts. The stream B contributes to the first join in both queries, and its timestamps are propagated to the subsequent WJ operation. Thus, the time constraints for tuples $b \in B$ are equivalent for both queries. We refer to Case $A1$ for more details and conclude that all output tuples of Q_1 are also detected in Q_2. □

Theorem 8. *SWJs of Case A4 are not associative in processing time and conditional associativity in event time.*

4.2 Session Window Joins

SessWJs create non-overlapping windows based on periods of tuple arrivals terminated by the gap parameter g. Thus, only Option (1) $c.ts \in C_j$ is relevant for SessWJs, where a tuple (a,b,c) is valid in Q_1 if $a.ts, b.ts \in w_i^1$, and $ab.ts, c.ts \in w_j^2$. Both streams contribute to a session, which may contain only a single tuple.
Case A5. We consider two equivalent session window operators, i.e., $W_1 = W_2$.

Processing Time. The time propagation between W_1 and W_2 is defined as follows: $\forall(a,b) : (a,b).ts = ts_e$.

Proof. Since all tuples $(a,b) \in w_i^1$ share a single timestamp, the sequence of c tuples primarily determines w_j^2 in Q_1. In contrast, w_j^2 in Q_2 is defined by $B \cup C$, i.e., the composition of unprocessed b and c tuples. Consequently, $w_j^2 \in Q_1 \neq w_j^2 \in Q_2$, leading to potentially different window lengths and, ultimately, divergent outputs. □

Event Time. We examine the propagation strategies $a.ts$ and $b.ts$:

Proof. **Propagation of A.ts.** $A \cup C$ defines w_j^2 in Q_1, whereas $B \cup C$ defines w_j^2 in Q_2. Consequently, $w_j^2 \in Q_1 \neq w_j^2 \in Q_2$, leading to discrepancies in the outputs of Q_1 and Q_2.
Propagation of B.ts. In Q_1 and Q_2, $B \cup C$ defines the window w_j^2. However, $b.ts$ in Q_1 is only propagated if b contributes to an output tuple (a,b). In contrast, the raw input of B is considered for the sessions in Q_2. For this reason, if a sequence of b tuples creates a session that contains no tuple $a \in A$, none of their timestamps are propagated for $w_j^2 \in Q_1$. Consequently, $w_j^2 \in Q_1 \neq w_j^2 \in Q_2$ and not all output tuples of Q_1 are also detected in Q_2. □

Theorem 9. *SessWJs are not associative.*

Consider the following support chat analytics scenario, which illustrates how the join order affects session boundaries and refutes Hypothesis 2 for SessWJs:

Example 4. *An online platform tracks customer interactions across multiple channels. To analyze complete conversations, session windows group events with a 5-minute inactivity gap. In query Q_1, customer messages (A) are first joined with agent responses (B), forming session $s_1^1 = (3,7)$ that captures two customer messages (a_3, a_7) and one agent response (b_3). The results of session s_1^1 are then joined with escalation tickets (C), creating the session $s_1^2 = (3,10)$, incorporating two tickets for further review (c_5, c_{10}). In query Q_2, agent responses are first joined with escalations, creating two sessions: $s_1^2 = (3,5)$ and $s_2^2 = (10)$ solely defined by the escalation c_{10}. As a result, s_2^2 reveals no output tuple for the join with customer messages. Thus, the escalation c_{10} is omitted, leading to an incomplete analysis of unresolved issues.*

4.3 Interval Joins

IVJs create a window w_n for each tuple t on the left join side, with interval bounds lB and uB relative to the timestamp of t. Since each t derives its substream from the right join side, IVJs are not logical clock aligned, duplicate-free, and only Option (1) $c \in C_j$ applies to form the output tuple (a,b,c). Thus, (a,b,c) is a valid output of Q_1 if $b.ts \in w_i^1 = [a.ts - lB, a.ts + uB]$ and $c.ts \in w_j^2 = [ab.ts - lB, ab.ts + uB]$. We examine both event time propagation strategies, a.ts and b.ts, as IVJs are exclusive to this domain.
Case A6. We consider equal-sized bounds for W_2,i.e.,$W_2.lB = W_2.uB$.

Proof. **Propagation of A.ts.** If $a.ts$ is propagated as the timestamp of AB, it defines the windows w_j^2 for the second join of AB with C in Q_1. In contrast, in Q_2, tuples $b \in B$ define the windows w_j^2 for the join with C. By definition, for (a,b,c) in Q_1, $b.ts \in [a.ts - lB, a.ts + uB]$. Let us probe the two corner cases for Q_2:
(1) $b.ts = a.ts - lB$: $\rightarrow w_j^2 = [a.ts - 2 \cdot lB, a.ts - lB + uB]$, where $a.ts - lB + uB = a.ts$. Thus, if c.ts > a.ts, the tuple (b,c) is not detected in $[B \bowtie C]^{W_2}$ and Q_2 is missing the result (a,b,c).
(2) $b.ts = a.ts + uB$: $\rightarrow w_j = [a.ts + uB - lB, a.ts2 \cdot uB]$, where $a.ts - lB + uB = a.ts$. Thus, if c.ts < a.ts, the tuple (b,c) is not detected in $[B \bowtie C]^{W_2}$ and Q_2 is missing the result (a,b,c).
Propagation of B.ts. If $b.ts$ is propagated as the timestamp of AB, it defines the windows $w_j^2 \rightarrow [b.ts - lB, b.ts + uB]$ for the join of AB with C in Q_1. In Q_2, each tuple b creates a window $w_j^2 = [b.ts - lB, b.ts + uB]$ for the join $[B \bowtie C]^{W_2}$, making w_j^2 equivalent in both queries. Thus, if (b,c) is detected in Q_1, it is also

detected in Q_2. For the second join $[BC \bowtie A]^{W_1}$, propagating $b.ts$ leverages commutativity between A and B (see Theorem 3). □

Theorem 10. *IVJs with equal-sized bounds are conditional associative.*

Case A7. We consider that the bounds of W_2 are not of equal size, i.e., $W_n.lB \neq W_n.uB$.

Proof (sketch). We apply the proof of Case $A6$ respectively to Case $A7$. In contrast to Case $A6$, for the propagation of $b.ts$, we refer to Theorem 4 that for unequal bounds IVJs are not commutative.

Theorem 11. *IVJs with unequal-sized bounds are not associative.*

5 ACQUISITION OF PROPERTIES

Our analysis of WJ properties reveals that WJ reordering is feasible without altering query semantics but remains limited across the full range of WJ types, depending on their semantics and the applied time propagation strategy. In this section, we introduce transformation rules that acquire the commutativity of Case $C4$ in Section 5.1 and expand the associativity of Case $A4$ in Section 5.2. In sum, these transformation rules enable reordering in five additional configurations. Additionally, we present a transformation rule for accurate time propagation in Section 5.3, facilitating reordering in conditionally associative cases while preserving query semantics. Then, we discuss the impact of time propagation in Section 5.3. Finally, we conclude our property analysis in Section 5.4.

5.1 Acquiring Commutativity for Case C4

The content-based windowing of IVJs causes non-commutativity when bounds are unequal, limiting alternative join orders. This issue is resolved by swapping bounds when reordering joins. Specifically, for $Q_1 = [A \bowtie B]^W$ with $W = (lB, uB)$, the reordered join $Q_2 = [B \bowtie A]^{W}$ must use $W' = (uB, lB)$. We revisit Case $C4$ and apply this boundary swap in Q_2:

Case C4 - Revisited. We consider the IVJ query Q_1 with unequal bounds $lB \neq uB$, which are swapped in Q_2. We then examine the two corner cases of Q_2, where tuples $b \in B$ define w_j, to verify whether all tuples (a, b) detected in Q_1 are also captured in Q_2.

Proof. (1) $b.ts = a.ts - lB$ in $Q_1 \rightarrow w_j \in Q_2 = [a.ts - lB - uB, a.ts - lB + lB]$. Thus, (a, b) is also detected in Q_2 as $a.ts \in [a.ts - lB - uB, a.ts]$.
(2) $b.ts = a.ts+uB$ in $Q_1 \rightarrow w_j \in Q_2 = [a.ts+uB-uB, a.ts+uB+lB]$. Thus, (a, b) is detected in Q_2 as $a.ts \in [a.ts, a.ts + lB + uB]$. □

Theorem 4 (Revisited). *IVJs with unequal-sized bounds are commutative if the bounds are swapped when the join order is reversed.*

Moreover, the boundary swapping also enables conditional associativity for Case $A7$.

Theorem 11 (Revisited). *IVJs with unequal-sized bounds are conditionally associative if their bounds are swapped whenever the join order is altered.*

5.2 Acquiring Full Associativity for Case A4

Our analysis reveals that equal-sized, non-overlapping SWJs exhibit commutativity and full associativity (see Section 4.1, Case $A2$).

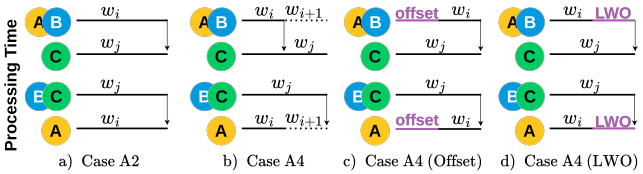


Figure 5: Approaches to Expand Associativity in Case A4.

This property arises because SPSs align window sequences created by W_n and W_{n+1} in multi-way WJs by the logical clock, ensuring $\forall w_i^n \exists w_j^{n+1} : w_i^n = w_j^{n+1}$. Figure 5a illustrates this alignment, regardless of the time domain, propagation strategy, or join order. In contrast, Case $A4$ loses this property due to differing window parameters (i.e., length and slide), resulting in misaligned windows. Figure 5b captures this misalignment, showing how altered join orders assign intermediate results to different windows, leading to inconsistent outputs. Adjusting window parameters and window assignment can restore full associativity in Case $A4$.

Processing Time. Aligned window ends ($t_{e_i} = t_{e_j}$) enable full associativity in Case A2 (see Figure 5a). Largest Window Only (LWO) [30] ensures alignment by applying the longest window length l_{max} to all window operators. The containment property guarantees that a reordered WJ query using LWO produces a superset of the original query Q, necessitating additional filtering for semantic equivalence. Furthermore, aligned window ends require all slide sizes to share a divisor relationship. However, LWO negatively impacts performance, as larger windows accumulate more data and increase computational load [25, 30]. We evaluate the impact of the window length in Section 7.

Event Time. The time propagation in event time spreads output tuples over the time interval of w_i. Thus, full associativity is achieved if each window boundary is either fully nested within or completely disjoint from the next. Formally, $w_i^n \subseteq w_j^{n+1} \vee w_i^n \cap w_j^{n+1} = \emptyset$. This alignment implies a divisor relationship between all window lengths in the query. We revisit the proof of Case $A4$ for a three-way SWJ with $W_n.l < W_{n+1}.l$, investigating the condition $w_i^n \subseteq w_j^{n+1}$ and the propagation of $a.ts$. In short, we ensure that any tuple (a, b, c) detected in $Q_1 = [[A \bowtie B]^{W_1}_{a.ts} \bowtie C]^{W_2}$ is also detected in $Q_2 = [A \bowtie [B \bowtie C]^{W_2}_{b.ts}]^{W_1}$. As Case $A4$ considers non-overlapping sliding windows, only Option (1) is of relevance, and the divisor relationship between windows guarantees that no partial overlap exists between w_i^1 and w_{j+n}^2. Formally, $\forall w_{j+n}^2, w_i^1 : w_{j+n}^2 \cap w_i^1 = \emptyset$ $(n \neq 0, n \in \mathbb{Z})$. Associativity for the propagation of $b.ts$ in Q_1 has been proven in Section 4.1.

Proof. **Propagation of A.ts.** (1) $c \in C_j \rightarrow c.ts \in w_j^2 \wedge (W_1.l \mid W_2.l \wedge W_1.l < W_2.l)$. It follows that $\forall b : b.ts \in w_i^1 \rightarrow b.ts \in w_j^2$. By applying the containment property [30], all (b, c) tuples of Q_1 are also in Q_2 ($w_j^2 \in Q_1 = w_j^2 \in Q_2$). However, since the time constraints on b tuples are relaxed due to $W_1.l < W_2.l$, Q_2 detects additional pairs (b, c) compared to Q_1. The first join $[B \bowtie C]^{W_2}$ of Q_2 has to propagate $b.ts$ to leverage commutativity (see Theorem 1). In the second join $[BC \bowtie A]^{W_2}$, this ensures that (b, c) tuples are filtered by aligning a and b tuples according to the window constraints of W_1, reducing the superset of $(b, c) \in Q_2$. Since $w_i^1 \in Q_1 = w_i^1 \in Q_2$, the output of both queries is equivalent. □

Algorithm 1 DeriveAllWindowPermutations(ω, W)

1: **for** $\bowtie_n \in \omega.keys$ **do** ▷ For each join operator in the function ω
2: **if** $\bowtie_n[0]$.size > 1 **then** ▷ Check if multiple streams are composed in the left join side
3: $\bowtie_n^k \leftarrow$ decomposeJoinPair($\bowtie_n$) ▷ If true, decompose, e.g., (AB:C) into (A:C), (B:C)
4: **for each** $\bowtie^k \in \bowtie_n^k$ **do** ▷ Check if left side is time propagating stream of current
5: $W_n \leftarrow \omega.get(\bowtie_n)$ ▷ Get the window specification for $\bowtie_n$
6: **if** $\bowtie^k[0] == W_{n-1}$.getTimeProgation() **then**
7: ω.add($\bowtie^k$, W_n) ▷ E.g., (A:C):W_2
8: **else** ▷ E.g., (B:C)
9: $windows \leftarrow W_x \in W : x \leq n$
10: ω.add($\bowtie^k$, windows) ▷ E.g., (B:C):W_1, W_2
11: $\omega, W' \leftarrow$ generateCommutativePairs(ω, W) ▷ incl. boundary swap for IVJs (see Section 5.1)
12: **return** ω, W' ▷ (A:B):{W_1}, (B:A):{W_1},(A:C):{W_2},(C:A):{W_2}, (B:C):{W_1, W_2},(C:B):{W_1, W_2}

We propose the transformation rule Largest Window First (LWF) to extend this property to configurations where $W_1.l > W_2.l$ and the more selective window operator W_2 excludes valid compositions of Q_1, if applied first in Q_2. To ensure semantic equivalence, LWF swaps the window operators to leverage the containment property of larger windows in the first join. In addition, LWF adjusts the time propagation from $b.ts$ to $c.ts$ to preserve commutativity for $A \bowtie C$, leading to $Q_2 = [A \bowtie [B \bowtie C]^{W_1}_{c.ts}]^{W_2}$.

Window Offsets and Their Limitations. Since window misalignment prevents full associativity, we examine the alignment strategy of offsets [23]. Offsets shift entire window sequences by adjusting logical clock references, i.e., modifying start times from t_b to t_b+ offset as shown in Figure 5c. While offsets must be preserved when present in the original query, they cannot acquire full associativity as they do not affect the divisor relationships between window operators. Following our findings, we revisit Theorem 8:

Theorem 8 (Revisited). *SWJs of Case A4 are fully associative in processing time under the condition that all window ends of reverted window operators W_n are aligned and $W_1.s \mid W_2.s$. SWJs of Case A4 are fully associative in event time under the condition that $W_1.l, W_1.s \mid W_2.l, W_2.s$ and the reverted join orders process all window assignments of the original query with larger windows first.*

5.3 Leveraging Time Propagation

Time propagation poses a unique challenge for WJs because each output tuple carries multiple timestamps to subsequent WJs. Thus, associativity is governed by the different propagation strategies.

Processing Time. A single timestamp, i.e., the maximal timestamp of each window w_i, is assigned to *all* output tuples, neglecting the timestamps of the contributing tuples. For this reason, associativity holds only for non-overlapping sliding windows with logical clock alignment, as observed in Case *A2* (see Section 5.2). For Cases *A1* and *A3* with overlapping windows, *shifting* all tuples to a single larger timestamp disrupts full associativity since intermediate results may fall into different windows when the join order changes.

Event Time. A contributing tuple's timestamp is propagated, which enables *conditional* associativity, analogous to key dependencies in relational joins. These dependencies do not necessarily degrade performance but do alter query semantics. In particular, as shown for Cases *A1*-*A4*, *A6*, and *A7* in Table 2, propagating the timestamps of stream B ($ET^*_{b.ts}$) preserves semantic equivalence, whereas propagating the timestamps of stream A ($ET^*_{a.ts}$) does not. Thus, WJs can exploit the dependencies in other join orders as long as the time-propagating streams are involved. For instance, in query $Q = [[A \bowtie B]^{W_1}_{a.ts} \bowtie C]^{W_2}$ that propagates $a.ts$, all join orders where A appears in the first join are semantically equivalent.

Table 2: Commutative and Associative Properties by Solution.

Case	Type	Parameter Setting	Time Domain $ET^*_{a.ts}$	Time Domain $ET^*_{b.ts}$	Time Domain PT
Commutative Properties					
C1	*SWJ*	-	✔	✔	✔
C2	*SessWJ*	-	✔	✔	✔
C3	*IVJ*	lB = uB	✔	✔	-
C4	*IVJ*	lB != uB	●(§ 5.1)	●(§ 5.1)	-
Associative Properties					
A1	*SWJ*	$W_1.l = W_2.l \wedge slide < length$	✗	✔(§ 5.3)	✗
A2	*SWJ*	$W_1.l = W_2.l \wedge slide \geq length$	✔	✔	✔
A3	*SWJ*	$W_1.l \neq W_2.l \wedge slide < length$	✗	✔(§ 5.3)	✗
A4	*SWJ*	$W_1.l \neq W_2.l \wedge slide \geq length$	●(§ 5.2)	✔(§ 5.3)	●(§ 5.2)
A5	*SessWJ*	gap g	✗	✗	✗
A6	*IVJ*	$W_n.lB = W_n.uB$	✗	✔(§ 5.3)	-
A7	*IVJ*	$W_n.lB \neq W_n.uB$	✗	●(§ 5.1, § 5.3)	-

$ET^*_{t.ts}$ = event time with user-defined time propagation of t, PT = processing time, ✔= semantically equivalent (default), ✗= changed semantics, ●= semantically equivalent (acquired)

To systematically derive valid join orders, Algorithm 1 extends the window assignment function ω by determining which window operator applies to which join pair, leveraging the predefined time propagation. It takes ω and the window specifications W (see Section 2) as input and iterates through all join pairs $\bowtie_n$. If a pair $\bowtie_n$ involves more than two streams (Line 2), it is decomposed into pairs $\bowtie_n^k$ containing a single stream per join side (Line 3). If the left side of a decomposed pair $\bowtie^k$ [0] is a time-propagating stream, then the operator W_n of $\bowtie_n$ is assigned to $\bowtie_n^k$, reflecting the dependency on its timestamp (Lines 6–7). Otherwise, the pair $\bowtie_n^k$ inherits all windows in Q relevant to its streams (Lines 8–10). Algorithm 1 returns the expanded mapping of ω, including updated window specifications for IVJs where applicable. For WJs classified as *conditional* associative, ω assigns the respective windows to all join pairs, and each pair that yields only a list of windows can be discarded. For instance, given Q, the join order $[[B \bowtie C]^{W_x} \bowtie A]^{W_y}$, yields the list $\{W_1, W_2\}$ as a possible assignment for W_x and can thus be discarded. In contrast, fully associative WJs of Cases *A2* and *A4* can exploit multiple window assignments: As proven in the revisit of Case *A4* (see Section 5.2), assigning the largest window of the list to W_x yields semantic equivalence. However, the join orders remain invalid if the list contains less than two sliding windows with $s < l$.

5.4 Classification of Window Join Properties

Our analysis, summarized in Table 2, reveals that WJ properties are governed by two key factors: window semantics and time propagation strategies. Based on these characteristics, we classify WJs into three hierarchical levels: (i) commutative-only, (ii) conditionally associative, and (iii) fully associative.

Commutative-only: The semantics of SessWJs restrict reordering to only two valid join orders, regardless of the time domain (Case *A6*). Additionally, overlapping SWJs in processing time (Case *A1*, *A3*) lack associativity due to the time propagation strategy of this time domain (Section 5.3). However, adapting event-time propagation strategies for intermediate results can restore associativity.

Conditional associative: SWJs and IVJs exhibit conditional associativity under event time, leveraging rule-based timestamp propagation (see Section 5.3) and boundary swapping (see Section 5.1). Alternative propagation strategies, e.g., using maximum or minimum timestamps [23, 34], still do not achieve full associativity, as they can also lead to the loss of intermediate results. Under these conditions, the search space expands to at most $2(n-1)!$ join orders.

Full associative: Non-overlapping SWJs (Cases *A*2 and *A*4) exhibit full associativity in both time domains, enabling maximal flexibility with $n!$ enumerated join orders (see Section 5.2).

Across all levels, associativity is maintained when timestamp propagation preserves all intermediate results across ancestor and successor windows and when window sequences remain unchanged.

6 *WJR* ALGORITHM AND INTEGRATION

We introduce our enumeration algorithm *WJR* in Section 6.1 and discuss its integration into existing SPSs in Section 6.2.

6.1 Window Join Enumeration Algorithm

Algorithm 2 presents *WJR*, our WJ enumeration algorithm that systematically generates semantically equivalent join orders for a given WJ query Q by assigning appropriate window operators and propagating timestamps for each join pair. As a first step, *WJR* extracts window-related details from Q, i.e., (1) the set of window specifications W that define the window type and parameters for each window operator W_n, and (2) the window assignments of ω, indicating which window W_n is assigned to which join pair $\bowtie_n$. Then, *WJR* adjusts and extends the window specifications W based on the time domain of Q and its window parameters. In particular, *WJR* checks and, if possible, applies LWO for processing time (Line 4, see Section 5.2), whereas it expands the initial window assignment ω to all possible join pairs in Q for event time (Line 8, see Section 5.3). Next, *WJR* generates all candidate join orders Q_{allJOs} and iterates through them (Lines 9-11). For each join $\bowtie_n$ in a candidate order Q_p, it verifies that a valid window assignment exists under our transformation rules and determines the time propagation attribute (Lines 12–14). If all $\bowtie_n$ are valid, the cost of Q_p is estimated using a simple cost model that accumulates the intermediate results of each join. Our cost model is based on a rate-based cost function [65], common for SPSs. We enhanced this function with window-specific factors based on our analysis of WJ semantics to better capture the impact of windowing on join costs. In particular, for a given WJ query Q, that joins n streams $S^k (k = 1, ..., n)$ using $n-1$ window operators $W_j (j = 1, ..., n-1)$, we estimate the cost as follows:

$$c(Q) = \sum_{j=1}^{n-1} \left[\left(\prod_{k=1}^{j+1} r(S^k) \right) \times \sigma_j \times \underbrace{\begin{cases} \left(\frac{W_j.l}{\Delta t} \right)^2 \cdot \frac{\Delta t}{W_j.s}, & \text{if } W_j \text{ is SW} \\ \frac{W_j.lB \ + \ W_j.uB}{\Delta t}, & \text{if } W_j \text{ is IV} \end{cases}}_{\omega(W_j) \text{ (window factors)}} \right]$$

, where $r(S^k)$ is the rate of S^k, and Δt is the base time granularity (e.g., seconds). The term $\frac{\Delta t}{W_j.s}$ penalizes overlaps by accounting for the number of windows triggered within Δt by SWJs, and $W_j.lB + W_j.uB$ represents the window length of W_j for IVJs. Valid join orders are collected and subsequently ranked in $Q_{validJOs}$ and then returned in Line 20. By enumerating WJ orderings, *WJR* can extend conventional search algorithms for join order optimization, such as Dynamic Programming [54], to support reordering in SPSs.

6.2 System Integration Details

In the following section, we outline practical considerations that enable the applicability of *WJR* within common SPSs.

Algorithm 2 *WJR(Q)*

```
1: ω, W ← getWindowSpecifications(Q)
2: if Q is in PT then
3:     if W contains SWJ && s ≥ l then
4:         W' ← checkAndApplyLWO(Q, W)                    ▷ see Section 5.2
5:     else
6:         return generateCommutativeJoinOrders(Q)
7: if Q is in ET then
8:     ∇ω, W' ← deriveAllWindowPermutations(ω, W)         ▷ see Section 5.3, Algorithm 1
9: Q_allJOs ← generateJoinOrders(Q)
10: Q_validJOs ← {}                                       ▷ Initialize result set to collect valid join orders
11: for Q_p in Q_allJOs do
12:     while Q_p.getNextJoinPair() do
13:         ⋈_n ← Q_p.getNextJoinPair()
14:         valid ← assignWindowOperator(⋈_n, ∇ω)
15:         if !valid then
16:             break
17:         if valid ∧ !Q_p.getNextJoinPair() then
18:             Q_p.cost ← estimateCost(Q_p)              ▷ Assign cost estimation for ranking
19:             Q_validJOs ← add(Q_p)                     ▷ Add valid join order, ranked by cost, to result set
20: return Q_validJOs                                     ▷ Return all valid join orders
```

Integration into Existing SPSs. *WJR* can be seamlessly integrated into the query optimizer of existing SPSs by incorporating our WJ transformation rules as logical rewriting rules. Many modern SPSs, including Apache Flink, Storm, and Beam, leverage rule-based optimization frameworks such as Apache Calcite [8]. In these systems, our transformation rules would be applied during the logical optimization phase. For instance, a new Calcite rule *WindowJoinReorderRule* would automatically generate alternative plans when matching query patterns are detected in a query plan.

By integrating *WJR*, SPSs can explore multiple semantically equivalent WJ plans instead of being constrained to a single fixed order. This unlocks optimization techniques that were previously unavailable or highly restricted. For instance, cost-based optimization, which is largely impractical without alternative plans, can now be leveraged to rank and select efficient join orders. Likewise, placement strategies, operator fusion, and sharing benefit from the increased flexibility due to the availability of multiple plans [31].

Handling Limited Stream Statistics. Cost models rely on statistics, such as stream rates or join cardinalities, which are challenging to obtain in SPSs compared to relational databases, making their collection an active research area [12, 49]. Because precise statistics are typically unavailable, heuristic approaches are essential. Without statistics, SPSs can leverage static window specifications to estimate the relative ranking of join orders heuristically. Specifically, our cost model in Section 6.1 incorporates key window parameters (e.g., length and slide size), enabling simple rankings even without exact stream rates. Additionally, systems can integrate user-provided hints (e.g., estimated or relative stream rates) to guide order selection [19]. Moreover, WJs are often combined with windowed aggregations that reduce input streams to a single tuple per group (key) and window. This property facilitates approximating the maximal arrival rate of aggregated output streams per key [34]. Together, these heuristics help guide the deployment of *WJR* when detailed statistics are unavailable.

Support for Multiple Window Types. Although our analysis focuses on queries with a single window type, *WJR* can handle WJs with multiple window types, e.g., a three-way WJ with the window operators $W_1 = I(lB, uB)$ and $W_2 = SW(l, s)$ [29]. This is possible because all WJs, except for SessWJs, are conditionally associative under event time, which underpins the foundation of our WJR algorithm, i.e., explicitly assigning each join pair the appropriate

Table 3: Summary of Queries

Query	Case	Window Specification		Intermediate Rates Per Key Partition							
				AB (W_1)			AC (W_2)			BC	
		W_1	W_2	**S1**	**S2**	**S3**	**S1**	**S2**	**S3**	**S1**	**S2**
Q_1	*A1*	$SW(20,2)$	$SW(20,2)$	750	1500	-	750	100	-	-	-
Q_2	*A1*	$SW(20,10)$	$SW(20,10)$	150	300	-	150	20	-	-	-
Q_3	*A3*	$SW(10,2)$	$SW(20,2)$	188	375	-	750	100	-	-	-
Q_4	*A3*	$SW(20,10)$	$SW(15,10)$	150	300	-	84	11	-	-	-
Q_5	*A2*	$SW(30,30)$	$SW(30,30)$	113	225	-	113	15	-	113	8
Q_6	*A2*	$SW(30,45)$	$SW(30,45)$	75	150	-	75	10	-	75	5
Q_7	*A4*	$SW(30,30)$	$SW(5,30)$	113	225	-	3	< 1	-	113	8
Q_8	*A4*	$SW(30,45)$	$SW(5,45)$	75	150	-	2	<1	-	75	5
Q_9	*A4*	$SW(5,30)$	$SW(30,30)$	3	6	-	113	15	-	113	8
Q_{10}	*A4*	$SW(5,45)$	$SW(30,45)$	2	4	-	75	10	-	75	5
Q_{11}	*A7*	$I(0,10)$	$I(10,10)$	38	75	-	75	10	-	-	-
Q_{12}	*A6*	$I(10,10)$	$I(10,10)$	75	150	-	75	10	-	-	-
Q_{13}	*A7*	$I(10,10)$	$I(10,0)$	75	150	-	38	5	-	-	-
Q_{14}	*A3, A4*	$SW(60,30)$	$SW(5,5)$	-	-	3000	-	-	<1	-	-
Q_{15}	*A3, A7*	$SW(30,15)$	$I(0,10)$	-	-	1500	-	-	3	-	-

window operators W_n (see Algorithm 1). As a result, *WJR* only needs to treat SessWJs and full associative WJs as a special case to manage heterogeneous window configurations correctly.

Overall, our implementation considerations support the integration of WJ reordering into existing SPS architectures, even when detailed statistics are unavailable and multiple window types are in use. By enabling WJ reordering, *WJR* broadens the range of optimization opportunities, improving both single-query and multi-query optimization scenarios under a variety of SPS constraints.

7 EVALUATION

In this section, we evaluate the impact of WJ reordering on performance and resource utilization. We describe our experimental setup in Section 7.1. In Section 7.2, we introduce our applied case validation, where we evaluate each theoretical case through actual queries. We investigate the impact of join reordering in Section 7.3.

7.1 Experimental Setup

In the remainder, we present hardware and software configurations as well as details about data and workloads.

Hardware and Software. We conduct our experiments on a two-node cluster. Each node is equipped with a 16-core Intel Xeon Silver CPU (4216, 2.10 GHz) and 528 GB of RAM. We use one node exclusively as a master and one as a worker. We use Apache Flink (version 1.11.6) for all experiments, as it is the only SPS that supports all four analyzed WJ types (see Section 2).

Metrics. We measure the maximum sustainable throughput (short: throughput) in tuples per second (tpl/s), representing the highest throughput the SPS can handle before latency increases and backpressure occurs. For each valid join order of a query Q_n, we assess this throughput and use it as the ingestion rate, ensuring that we evaluate each query at its maximum feasible rate. We report speedup as the throughput ratio between the best and worst join order. To prevent the sink from becoming a bottleneck, we ensure a constant output rate across all queries and collect latency, i.e., the time from tuple creation (ts_{PT}) to its arrival at the sink [36], per second. Additionally, we measure CPU and memory usage.

Workloads. *Data.* We use two real-world sensor datasets for our applied case validation, with samples available on our GitHub repository (1): (1) traffic congestion data (vehicle quantity and velocity) collected per minute, and (2) air quality data [55] (particulate matter, temperature, humidity) collected every 3–5 minutes.

For performance analysis, we generate synthetic workloads using a parallel source function that uniformly distributes tuples of the form (id : int, $value$: int, ts_{ET} : long, ts_{PT} : long) across 16 key partitions p (one per core). Each stream S is generated by a dedicated source instance running for 25 minutes [36]. The ingestion rate for each stream is determined by the evaluated throughput of the join order, proportioned based on event-time arrival rates.

Event Time Arrival Rates. Event time arrival rates define the number of tuples occurring within a window and, thus, influence join selectivity. For our evaluation, we use three baseline settings: *S1*: r(A)=r(B)= r(C)=15; *S2*: r(A)=30, r(B)=15, r(C)=1; and *S3*: r(A)=15, r(B)=100, r(C)=1. Here, $r(S^p)$ denotes the arrival rate within 60 time units for each key partition p of stream S ($p = 1, ..., 16$). We use $r(S^p)$ for simplicity, as all partitions produce the same rate. Low-rate streams may not appear in every window (e.g., r(S)=1 with $SW(30,30)$), resulting in fewer join matches and highly selective intermediate results. In contrast, higher-rate streams (e.g., r(B)=100) increase intermediate results (see also Table 3). To ensure the system processes tuples at its tested throughput capacity, we scale the actual ingestion rate proportionally to these arrival-rate settings.

Queries. We define the baseline queries $Q_1 - Q_{15}$, each corresponding to a three-way WJ $Q_n = [[A \bowtie B]^{W_1} \bowtie C]^{W_2}$ and execute all their valid join orders. Each query represents an associative case from Table 2, excluding SessWJs. To systematically assess the impact of window parameters, we introduce variations per WJ type. Table 3 details query specifications and shows the estimated intermediate arrival rates of a key partition for each setting, computed using our cost function in Section 6.1 ($\sigma_j = 1$). Additional queries and settings are introduced in the respective evaluation sections. Unless stated otherwise, all queries exclusively contain WJs.

7.2 Applied Case Validation

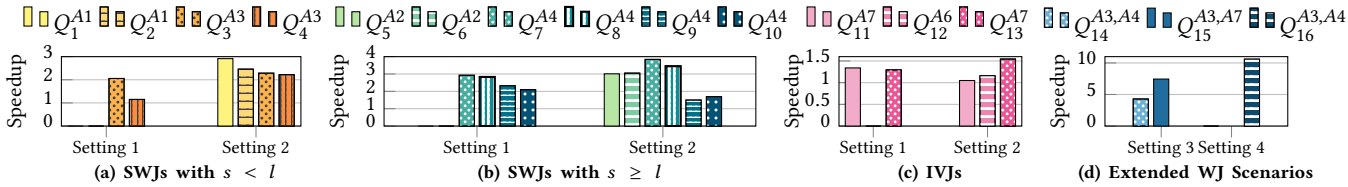
To underscore the correctness and completeness of our case study and proposed transformation rules, we provide various example queries in our GitHub repositories covering all commutativity and associativity cases in event time outlined in Section 3 and 4, as well as cases involving more than two joins. Each case is implemented as a JUnit test, running different join orders to demonstrate how commutativity and associativity properties (or their absence) affect query outcomes using both small synthetic datasets and real-world samples from *QnV* and *AQ-Data*, covering diverse scenarios.

7.3 Impact of Window Join Reordering

In this experiment, we evaluate the impact of reordering WJs based on our findings, which enable multiple join orders instead of a fixed plan per query. We omit figures for queries with identical rates and window operators since their results are equivalent. The following subsections detail the outcomes for each WJ type.

7.3.1 Sliding Window Joins with $s < l$. Workloads. We evaluate Case A1 ($W_1 = W_2$) and Case A3 ($W_1 \neq W_2$) under Setting 1 and 2 with slide parameters $s = 2$ and $s = 10$, i.e., $Q_1 - Q_4$. Each query permits four orders due to *conditional* associativity (see Section 5.3).

Observations. In Figure 6a, we present the speedups observed for $Q_1 - Q_4$. First, the throughput is very low, i.e., below 200,000 tpl/s for $s = 2$ and 2 million tpl/s for $s = 10$. This is because overlapping SWJs generate duplicates. In particular, the smaller the slide, the



(a) SWJs with $s < l$ (b) SWJs with $s \geq l$ (c) IVJs (d) Extended WJ Scenarios

Figure 6: Speedup of Maximum Sustainable Throughput between Best and Worst Join Order.

more duplicates are in the intermediate results. These duplicates must be maintained in the operator state, which increases memory usage and garbage collection overhead, resulting in lower throughput. The high intermediate result rates shown in Table 3 reflect the heavy computational effort caused by these duplicates. Second, commutative pairs have similar throughput, with an average speedup of 1.09 (min. 1.01, max. 1.15). Third, associative pairs yield significant performance gains. In Setting 1, where streams have equal input ratios, the different window lengths determine the best and worst join orders with speedups of up to 2 (avg. 1.52, min. 1.13). In Setting 2, throughput improves with larger slides, and speedups between join orders reach up to 2.91 (min. 1.98, avg. 2.35).

7.3.2 Sliding Window Joins with $s \geq l$. Workloads. We evaluate Case *A*2 ($W_1 = W_2$) and the two configurations of Case *A*4 identified in Section 5.2 ($W_1.l > W_2.l$ and $W_1.l < W_2.l$) under Setting 1 and 2 with slide parameters $s = l$ and $s > l$, i.e., $Q_5 - Q_{10}$. All queries permit six join orders due to full associativity.

Observations. In Figure 6b, we show the speedups observed for $Q_5 - Q_{10}$. First, non-overlapping SWJs achieve significantly higher throughput than overlapping ones, ranging from 1.2 million tpl/s (lowest for Q_5) to 9.55 million tpl/s (highest of Q_{10}). In particular, the throughput increases with the slide size. Second, commutative pairs have similar throughput across all cases, with an average speedup of 1.08 (min. 1.01, max. 1.19). Third, associative join orders yield substantial performance improvements. In Case *A*2, we observe speedups of up to 3 (min. 2.52, avg. 2.83). For Case *A*4, when $W_1.l > W_2.l$ (Q_7 and Q_8), speedups reach up to 3.84 (min. 2.83, avg. 3.26), whereas when $W_1.l < W_2.l$ (Q_9 and Q_{10}), speedups are lower, up to 2.33 (min. 1.5, avg. 1.9). The main reason for the different speedup results is that Q_7 and Q_8 execute the expensive join between *A* and *B* over the larger window operator W_1, while Q_9 and Q_{10} do so over the smaller window operator W_2. Thus, Q_7 and Q_8 profit more from reordering, where our transformation rules enable the processing of the more restrictive (i.e., selective) window operators first, to reduce intermediate results and achieve higher throughput.

7.3.3 Interval Joins. Workloads. We evaluate Case *A*6 with equal-sized, and Case *A*7 with unequal-sized boundaries and boundary swapping (Section 5.1) under Setting 1 and 2, i.e., $Q_{11} - Q_{13}$. Each query permits four join orders due to *conditional* associativity.

Observations. In Figure 6c, we present the speedups observed for $Q_{11} - Q_{13}$. First, IVJs exhibit distinct throughput behavior. At ingestion rates that avoid prolonged backpressure, latencies remain in the order of minutes, a phenomenon not seen with SWJs. When the ingestion rate is reduced to achieve latencies of seconds, throughput drops below 400,000 tpl/s, comparable to overlapping SWJs with small slide values. Second, commutative pairs yield similar

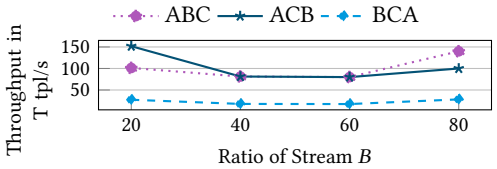


Figure 7: Impact of Tuple Distribution on Throughput

throughput across all cases, with an average speedup of 1.02 (min. 1.01, max. 1.05), indicating that boundary swapping does not affect performance. Third, associative pairs have a more pronounced effect, achieving speedups of up to 1.5 (min. 1.05, avg. 1.34).

7.3.4 Extended Window Join Scenarios. Workloads. We run three queries that extend the base case by containing multiple window types and more streams. In particular, Q_{14} and Q_{16} mix overlapping and non-overlapping sliding windows. Furthermore, Q_{16} is a five-way join with the following window specifications: $W_1 = W_2 = SW(20, 10)$, $W_3 = SW(20, 20)$, and $W_4 = (20, 5)$, and arrival-rate setting 4: $r(A) = 29$, $r(B) = 2$, $r(C) = 8$, $r(D) = 1$, and $r(E) = 60$. Q_{15} contains one overlapping sliding window and an IVJ. We apply a positive integer filter to the *value* attribute (random Integer) across all streams, disrupting their uniform distribution.

Observations. In Figure 6d, we show the speedups achieved for these three queries. While commutative combinations yield similar throughput consistent with previous observations, associative combinations involving multiple windows and non-uniform distributions show more pronounced speedups. Specifically, we observe speedups of up to 10.6 (min. 4.22, avg. 7.27) when costlier overlapping sliding windows are executed after smaller, non-overlapping window operators. The throughput ranges from 4,000 tpl/s (lowest for Q_{16}) to 7.9 million tpl/s (highest of Q_{14}).

7.3.5 Impact of Rate Distribution. Workloads. We run Q_5 under different event-time arrival rates, i.e., we fix $r(A) = 1$, vary $r(B)$ (e.g., 20, 40, etc.), and assign the remaining tuples to stream C to maintain a total of 100. Since commutative pairs yield similar results, we focus on the join orders ABC, ACB, and BCA.

Observation. In Figure 7, we present the throughputs of the three join orders. First, the orders ABC and ACB exhibit symmetric performance: ACB outperforms when $r(B)$ is low, while ABC is preferable when $r(B)$ is high, i.e., processing the more frequent stream first with *A* prevents an unnecessarily large intermediate result. Second, BCA consistently achieves the lowest throughput across all settings because it executes the costliest join first. The maximum observed speedups across the tested settings range from 4.6 to 5.5.

7.3.6 Resource Utilization and Latency. At *maximum sustainable throughput*, *memory usage* is the primary bottleneck: All join orders fully utilize available memory, while *CPU utilization* stays

moderate at 34% for IVJs and 57% for SWJs. Thus, speedups are attributed to join orders that reduce intermediate results. Orders with extensive intermediate results require more CPU resources and occasionally cause garbage collection stalls. *Latency* likewise remains low: SWJs average 3 s (min. 0.4 s, max. 1 min), and IVJs average 1 s (min. 0.1 s, max. 5 min). The upper ranges occur occasionally when garbage collection of large operator states interrupts the execution pipelines.

7.4 Observations on Join Order Performance

Our experiments confirm that the join order strongly impacts WJ performance. Since the most complex WJ typically determines the overall pipeline throughput, suboptimal orders create performance bottlenecks [31]. A key takeaway of our evaluation is that placing more selective joins with lower arrival rates and restrictive window parameters upfront improves execution efficiency. Examples of such parameters include smaller window lengths (Q_3), larger slide sizes (Q_8), or duplicate-free windows (Q_{15}). Starting the execution pipeline with highly selective joins minimizes intermediate results and unnecessary computations in downstream operations, which improves performance by decreasing memory usage and avoiding garbage collection stalls. Additionally, our evaluation indicates that queries with high variance in stream rates ($S2$) or window parameters (Q_7) benefit the most from reordering. Moreover, our rate-based cost function (Section 6.1), enhanced with window-specific factors, accurately reflects WJ performance, i.e., inefficient orders correlate with higher costs due to large intermediate results. These findings align with the principles of relational database optimizers, which also prioritize join orders that minimize intermediate results, as the query cost is directly tied to them [17, 32, 41]. An SPS optimizer could leverage these takeaways by applying and enhancing our simple cost model to rank and prune WJ orders. Even without stream rates, our insights suggest that static window properties and user-provided heuristics can guide order selection (see Section 6.2).

8 RELATED WORK

Stream Processing Optimizations. Most stream processing optimizations are rule-based and are applied statically at compile-time [31, 58]. Query plan optimizations consider operator reordering, separation, fusion, and fission [31]. Techniques proposed for reordering focus on stateless operations such as selection and projection [6], which improve performance by minimizing the number of tuples and their size early in the execution process [31, 34, 58]. However, WJ optimizations remain confined to techniques that preserve the query graph, such as load shedding or algorithm selection [26, 36, 48] even though preliminary studies show promising results [58]. Moreover, operator reordering is a key enabler of other stream processing optimizations [31], such as redundancy elimination, placement, or multi-query merging [10], all of which enhance the performance of complex stream queries but are currently not possible for MWJs. Our work enables WJ reordering, thus providing the foundation for advanced WJ optimizations in SPS.

Current research on stateful operations involving windowing has mostly centered around optimizations for window aggregation [9, 61, 64] with less focus on WJs [18, 37]. Those optimizations target the improvement of the actual execution instead of the execution pipeline. Thus, they leave the query graph unchanged but utilize operator placement [44], load shedding [26], algorithm selection [48], incremental computation, slicing, and resource sharing [37, 61], as well as tuple routing and smart data partitioning [18]. These optimizations enhance the efficiency of SPS and are orthogonal to our work in optimizing the query graph.

Another related line of research explores adaptive stream processing, where the SPS reacts to changes in the data characteristics to maintain performance for continuous queries [12, 18, 19, 28, 49, 58]. These approaches focus on dynamically collecting and maintaining stream statistics to estimate costs at runtime. Combining WJ enumeration with adaptive runtime statistics complements our approach towards cost-based join order selection.

Relational Join Query Optimization. Join query optimization is a well-studied field in the context of relational databases [41, 46]. Relational join query optimization aims to find the cheapest plan from all semantically equivalent plans for an input query to execute. To this end, optimizers use mathematical cost models based on factors like tuple counts, join selectivity, and intermediate result sizes. Algebraic rules from relational algebra allow query plans to be reordered while obtaining the same result [40, 42, 54]. However, traditional join optimization techniques do not account for the unique properties of WJs. Simply treating timestamps as ordinary attributes overlooks crucial aspects of streaming, such as out-of-order events and dynamic window boundaries. Consequently, WJs incorporate additional constraints, preventing the seamless applicability of relational join reordering techniques. Using our in-depth analysis of WJ semantics, we unlock the usage of relation join optimization for WJs in SPSs.

9 CONCLUSION

In this paper, we address the fundamental problem of WJ reordering in SPSs. By analyzing the algebraic properties of SWJs, SessWJs, and IVJs under both event time and processing time, we reveal that WJs diverge from relational join properties due to window semantics and time propagation strategies. To overcome these limitations, we propose three transformation rules that accommodate these constraints and enable WJ reordering without compromising query correctness. We consolidate these rules into our enumeration algorithm *WJR*. Our evaluation shows speedups of up to 10x across the different WJ types, underscoring the impact of join order on throughput and resource efficiency. As a result, our findings lay the foundation for more flexible and efficient query optimization in streaming scenarios by providing multiple alternative WJ plans, rather than a single one. A natural next step is to integrate our approach into the logical rewrite phase of an SPS optimizer, where WJ reordering can be leveraged alongside system-wide, statistics-independent objectives such as maximizing operator sharing across concurrent queries or strategically placing operators in decentralized deployments to reduce network overhead.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grant BIFOLD25B. We thank Philipp M. Grulich, Sebastian Bress, Leon Papke, and our reviewers for their insightful suggestions and comments.

REFERENCES

[1] Microsoft 2024. 2024. Microsoft Learn - Stream Analytics Query Language. https://learn.microsoft.com/en-us/stream-analytics-query/stream-analytics-query-language-reference. Accessed Sept. 2024.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[3] T. Akidau, S. Chernyak, and R. Lax. 2018. *Streaming Systems: The What, Where, When, and how of Large-scale Data Processing*. O'Reilly. https://books.google.de/books?id=48-BAQAACAAJ

[4] Samira Akili and Matthias Weidlich. 2021. MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 10–22.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers (Lecture Notes in Computer Science)*, Georg Lausen and Dan Suciu (Eds.), Vol. 2921. Springer, 1–19.

[7] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1757–1772.

[8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 221–230.

[9] Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. 2020. Disco: Efficient Distributed Window Aggregation. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 423–426.

[10] Ankit Chaudhary, Steffen Zeuch, Volker Markl, and Jeyhun Karimov. 2023. Incremental Stream Query Merging. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 604–617.

[11] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, Alberto O. Mendelzon and Jan Paredaens (Eds.). ACM Press, 34–43.

[12] Grigorios Chrysos, Odysseas Papapetrou, Dionisios N. Pnevmatikatos, Apostolos Dollas, and Minos N. Garofalakis. 2019. Data Stream Statistics Over Sliding Windows: How to Summarize 150 Million Updates Per Second on a Single Node. In *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*, Ioannis Sourdis, Christos-Savvas Bouganis, Carlos Álvarez, Leonel Antonio Toledo Díaz, Pedro Valero-Lara, and Xavier Martorell (Eds.). IEEE, 278–285.

[13] E. F. Codd. 1983. A Relational Model of Data for Large Shared Data Banks (Reprint). *Commun. ACM* (1983).

[14] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate Join Processing Over Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 40–51.

[15] Databricks. 2025. Virgin Australia boosts productivity, real-time data with the Databricks Data Intelligence Platform. https://www.youtube.com/watch?v=pWqx1P1BV1o. Accessed Feb. 2025.

[16] dbt Labs. 2025. dbt Case Studies. https://www.getdbt.com/case-studies. Accessed Jan. 2025.

[17] Bailu Ding, Vivek Narasayya, Surajit Chaudhuri, et al. 2024. Extensible Query Optimizers in Practice. *Foundations and Trends® in Databases* 14, 3-4 (2024), 186–402.

[18] Manuel Dossinger and Sebastian Michel. 2021. Optimizing Multiple Multi-Way Stream Joins. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1985–1990.

[19] Manuel Dossinger, Sebastian Michel, and Constantin Roudsarabi. 2019. CLASH: A High-Level Abstraction for Optimized, Multi-Way Stream Joins over Apache Storm. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1897–1900.

[20] Tarek Elsaleh, Shirin Enshaeifar, Roonak Rezvani, Sahr Thomas Acton, Valentinas Janeiko, and María Bermúdez-Edo. 2020. IoT-Stream: A Lightweight Ontology for Internet of Things Data Streams and Its Use with Data Analytics and Event Detection Services. *Sensors* 20, 4 (2020), 953.

[21] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Kamp, and Michael Mock. 2017. Issues in complex event processing: Status and prospects in the Big Data era. *J. Syst. Softw.* 127 (2017), 217–236.

[22] Apache Software Foundation. 2024. Apache Beam. https://beam.apache.org. Accessed Jun. 2024.

[23] Apache Software Foundation. 2024. Apache Flink. https://flink.apache.org. Accessed Feb. 25.

[24] Apache Software Foundation. 2024. Apache Spark. https://spark.apache.org. Accessed Jun. 2024.

[25] Apache Software Foundation. 2024. Apache Storm. https://storm.apache.org. Accessed Aug. 2024.

[26] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. 2007. A Load Shedding Framework and Optimizations for M-way Windowed Stream Joins. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 536–545.

[27] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352.

[28] Philipp M. Grulich, Sebastian Breß, 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2487–2503.

[29] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM 2003), 9-11 July 2003, Cambridge, MA, USA*. IEEE Computer Society, 75–84.

[30] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Scheduling for shared window joins over data streams. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, 297–308.

[31] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4 (2013), 46:1–46:34.

[32] Alexander K. Hudek, David Toman, and Grant E. Weddell. 2015. On Enumerating Query Plans Using Analytic Tableau. In *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24, 2015. Proceedings (Lecture Notes in Computer Science)*, Hans de Nivelle (Ed.), Vol. 9323. Springer, 339–354.

[33] Yannis E. Ioannidis. 1996. Query Optimization. *ACM Comput. Surv.* 28, 1 (1996), 121–123.

[34] Anand Jayarajan, Wei Zhao, Yudi Sun, and Gennady Pekhimenko. 2023. TiLT: A Time-Centric Approach for Stream Query Optimization and Parallelization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 818–832.

[35] Hyunchul Kang. 2013. In-Network Processing of Joins in Wireless Sensor Networks. *Sensors* 13, 3 (2013), 3358–3393.

[36] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1507–1518.

[37] 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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 607–622.

[38] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proc. VLDB Endow.* 11, 11 (2018), 1346–1359.

[39] Ilya Kolchinsky and Assaf Schuster. 2018. Join query optimization techniques for complex event processing applications. *Proc. VLDB Endow.* (2018).

[40] Donald Kossmann and Konrad Stocker. 2000. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.* 25, 1

(2000), 43–82.
[41] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
[42] Volker Markl, Guy M. Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Syst. J.* 42, 1 (2003), 98–106.
[43] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1078–1092.
[44] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. 2019. Efficient Operator Placement for Distributed Data Stream Processing Applications. *IEEE Trans. Parallel Distributed Syst.* 30, 8 (2019), 1753–1767.
[45] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (1991), 513–534.
[46] Thomas Neumann. 2018. Query Optimization (in Relational Databases). In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer.
[47] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (LNI)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.), Vol. P-265. GI, 31–50.
[48] Dwi P. A. Nugroho, Philipp M. Grulich, Steffen Zeuch, Clemens Lutz, Stefano Bortoli, and Volker Markl. 2024. Benchmarking Stream Join Algorithms on GPUs: A Framework and its Application to the State-of-the-art. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 188–200.
[49] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. 2015. Sketching distributed sliding-window data streams. *VLDB J.* 24, 3 (2015), 345–368.
[50] Steven Purtzel, Samira Akili, and Matthias Weidlich. 2022. Predicate-based push-pull communication for distributed CEP. In *16th ACM International Conference on Distributed and Event-based Systems, DEBS 2022, Copenhagen, Denmark, June 27 - 30, 2022*, Yongluan Zhou, Panos K. Chrysanthis, Vincenzo Gulisano, and Eleni Tzirita Zacharatou (Eds.). ACM, 31–42.
[51] Quix. 2024. A Guide to Windowing in Stream Processing. https://quix.io/blog/windowing-stream-processing-guide?utm_source=chatgpt.com Accessed Jan. 2025.
[52] Redpanda. 2023. Popular Stream Processing Patterns. https://www.redpanda.com/blog/popular-stream-processing-patterns?utm_source=chatgpt.com Accessed Jan. 2025.
[53] 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.
[54] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34.
[55] SENSOR.COMMUNITY. 2022. Global Sensornetzwork. https://sensor.community/de/ Accessed July 2024.
[56] Shobhit Seth. 2023. Basics of Algorithmic Trading: Concepts and Examples. https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp Accessed Jan. 2025.
[57] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2523–2537.
[58] Darya Sharkova, Alexander Chernokoz, Artem Trofimov, Nikita Sokolov, Ekaterina Gorshkova, Igor Kuralenok, and Boris Novikov. 2021. Adaptive SQL Query Optimization in Distributed Stream Processing: A Preliminary Study. In *International Workshop on Software Foundations for Data Interoperability*. Springer, 96–109.
[59] Snowflake. 2024. The Modern Data Streaming Pipeline. https://www.snowflake.com/wp-content/uploads/2024/03/The-Modern-Data-Streaming-Pipeline-1.pdf Accessed Jan. 2025.
[60] Tri Minh Tran and Byung Suk Lee. 2010. Distributed stream join query processing with semijoins. *Distributed Parallel Databases* 27, 3 (2010), 211–254.
[61] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 97–108.
[62] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. Nexmark–a benchmark for queries over data streams (draft). *Technical report* (2008).
[63] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2022. Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems. *Datenbank-Spektrum* 22, 2 (2022), 99–107.
[64] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011.
[65] Stratis Viglas and Jeffrey F. Naughton. 2002. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 37–48.
[66] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[67] Steffen Zeuch, Xenofon Chatziliadis, Ankit Chaudhary, Dimitrios Giouroukis, Philipp M. Grulich, Dwi Prasetyo Adi Nugroho, Ariane Ziehn, and Volker Markl. 2022. NebulaStream: Data Management for the Internet of Things. *Datenbank-Spektrum* 22, 2 (2022), 131–141.
[68] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.