SPO-Join: Efficient Stream Inequality Join

Adeel Aslam
University of Modena and Reggio Emilia, Italy
adeel.aslam@unimore.it

Kaustubh Beedkar
Indian Institute of Technology Delhi
India
kbeedkar@cse.iitd.ac.in

Giovanni Simonini
University of Modena and Reggio Emilia, Italy
giovanni.simonini@unimore.it

ABSTRACT

Stream inequality join aims to combine tuples coming from different streams based on inequality conditions and is a fundamental operator in distributed data stream processing. It is known to be computationally expensive as indexing data structures for determining matching tuples must be continuously updated.

To significantly alleviate this problem, we propose SPO-Join, a novel solution that combines a mutable B$^+$-tree for efficient insertions and an immutable sorted-array-based data structure for efficient searching. Furthermore, our proposed method is designed to be efficiently executed with distributed stream processing engines. Our experiments on real-world and synthesized datasets suggest that the proposed SPO-Join exhibits superior performance compared to state-of-the-art index-based stream inequality join solutions.

1 INTRODUCTION

Increase in the requirement for real-time data analytics has led to the massive adoption of distributed stream processing engines (DSPEs)—such as Storm[1], Flink[2], and Spark Streaming[3] —in many domains including energy, traffic, and health. Applications in such domains, for example, continuous monitoring of energy consumption, real-time traffic, and health data analysis, often involve joining data from multiple streams of data sources. Efficient stream joins, therefore, play an important role.

In the stream join, the goal is to find matching tuples from different data streams that satisfy a certain *join predicate*. Stream joins operations are well known to be computationally expensive, primarily because of the need to maintain continuously and update a data structure employed to facilitate finding the matching tuples [25]. To this end, several distributed stream join models and algorithms have been proposed [6, 12, 18, 35]. In the aforementioned works on stream joins, the focus has mainly been on *equality stream joins*, i.e., where the join predicate involves an equality condition on attributes of matching tuples. However, several applications require an inequality condition to hold for joining data streams. Consider the following example in the context of real-time energy consumption monitoring.

Example 1. *Consider CloudPro, a company that runs two data centers R and S. Both data centers have a similar cooling infrastructure but R is smaller than S in terms of number of servers and racks. CloudPro has to routinely reschedule jobs between its two data centers based on the power consumption from racks (POWER) and cooling units (COOL). For this, they analyze the real-time power consumption data every 10 minutes over a 60-minute window where R's rack power consumption is less R.POWER < S.POWER but cooling power usage is higher than S R.COOL > S.COOL.*

Q 1: Real-time data centers power consumption

```
SELECT R.POW_ID, R.COOL_ID, S.POW_ID, S.COOL_ID
FROM R, S
WHERE R.POWER<S.POWER AND R.COOL>S.COOL
WINDOW AS (SLIDE INTERVAL '10' ON '60')
```

Unlike a traditional stream join, the above example query involves joining data streams with inequality conditions. We refer to such joins as *streaming inequality join* or stream inequality join, which is common in applications. To gain a better understanding of the inequality join operator, let us consider another example with band join condition (band join contains inequality operator that returns all pairs of tuples that are close to each other [17]).

Example 2. *A transportation analyst needs to have access to real-time analysis of areas that have high demand for taxi services. By identifying clusters of trips in such areas, analysts can gain insights into traffic patterns, congestion hot spots, and peak travel times. To achieve this, the analyst has initiated a query to find taxi trips where the passenger's pickup locations (longitude (LON) and latitude (LAT)) are close to each other within a specific interval of time.*

Q 2: Real-time analysis of taxi trips

```
SELECT tripId, time FROM taxi_trips
WHERE ABS(start_LON1 - start_LON2) < 0.03
AND   ABS(start_LAT1 - start_LAT2) < 0.03
WINDOW AS (SLIDE INTERVAL 'D' ON 'W')
```

State-of-the-art [11, 18, 25, 35] distributed stream join approaches consist of two distinct levels of processing: the *router* and the *joiner* as depicted by Figure 1. A new tuple from any stream undergoes initial processing within the *router* component. Here, tasks like segmenting the tuple into distinct fields of query relation take place. Subsequently, the processed tuples are forwarded downstream toward the *processing elements* (*PEs*) of *joiner* component where indexes are employed to hold the contents of the streaming window for join operators. Each tuple is indexed or probed against these data structures (In Figure 1, each circle shows a processing element of DSPS for stream join components).

Existing stream join techniques differ in their choice of the data structure used for the indexing, such as B$^+$tree and CSS-tree [21, 25]. B$^+$tree is a type of self-balancing data structure that is widely used for indexing data items and retrieving them efficiently, especially for highly selective queries. However, this indexing solution is better suited for small-size sliding windows. If the window size increases, it introduces a high overhead for index updating and removal from the data structure. A CSS-tree indexing solution is employed for range queries, which is slightly better than B$^+$tree because it eliminates several pointers and uses relative indexing. However, inserting data items requires a lot of

[1] https://storm.apache.org/
[2] https://flink.apache.org/
[3] https://spark.apache.org/

Series ISSN: 2367-2005

10.48786/edbt.2025.12

reconstruction of indexing due to the implicit addresses of child pointers.

Despite this, many solutions employ these indexing data structures for inequality queries, which include chain index [18] that comprises several linked sub-indexes of B^+tree for the sliding window. Sub-indexes are conceptually divided into active and archive sub-indexes, where a new tuple can only be inserted into the active sub-indexes. However, it requires searching all sub-indexes among distributed processing elements of *joiner* component. Similarly, PIM-tree [25] divides the sliding window into a mutable linked set of B^+tree and a search-efficient immutable CSS tree data structure. A new tuple first explores the CSS tree to the depth d and then inserts it into the linked B^+tree indexing data structure that is pointed by the node of the CSS tree at depth d. Similarly, probing both mutable and immutable designs for query execution is required.

While tree-based indexes are a natural choice for inequality joins, Khayyat et al. [13] show that their IE-Join approach based on permutation and offset arrays yield better performance for fixed data. Yet, the IE-Join approach does not lend itself to streaming data. For instance, we empirically observed the IE-Join algorithm performance on a synthesized dataset for query Q1 with a match rate of 250 million tuples and found that it consumes 5.3×, 4.65×, and 21.25× less computation time than the B^+tree indexing, CSS tree indexing, and naive nested-loop join algorithm, respectively. However, for stream processing, the sliding window is continually updated with the arrival of new tuples, posing challenges to maintaining a sorted order of slide data items.

In this paper, we propose a new approach for distributed stream inequality join based on the two-tier data structure [18, 25]. The *PEs* of the *joiner* component hold two distinct structures: an insert efficient mutable B^+tree data structure and search efficient immutable structure called permutation and offset-based join (PO-Join). This whole join strategy is named as *stream permutation and offset-based join* (SPO-Join) [4]. Most of the sliding window contents are held by the PO-Join structure. To ensure completeness, each new tuple is evaluated against both in-memory data structures; however, it is only inserted into the mutable part of SPO-Join.

This paper is organized as follows: Section 2 discusses preliminaries and related work about stream join. Moreover, it also discusses the challenges associated with adopting SPO-Join. Section 3 provides the overview of the proposed SPO-Join strategy and a detailed discussion of distributed SPO-Join; Section 4 provides a discussion on different aspects of efficient distributed query processing. In Section 5, we describe our experiments. Section 6 concludes the paper.

2 BACKGROUND

This section provides preliminary details and related work about stream join. Finally, we discuss the challenges with the SPO-Join.

2.1 Preliminaries

Streaming data. Continuous data comprises a sequence of tuples $T = \{t_1, t_2, t_3, \ldots\}$ with undefined input data rate. A tuple t_i consists of a key $k_i \in \{1, n\}$ as a data item and its payload as $\{v_i \in V\}$; where V is a real value that represents any possible value within a specified domain $V \in \mathbb{R}$. The tuple is represented as $t_i =< k_i, v_i >$.

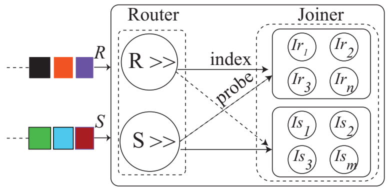


Figure 1: Distributed stream join for two distinct streams

Sliding window. The bounded set of streaming tuples is termed as sliding window $W_L = \{t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+L}\}$; where L is the total items in the sliding window. It can be classified into two primary categories: count-based W_c or time-based W_t tuple boundaries. On the arrival of a new tuple t_i, the obsolete tuples are discarded from the predefined size of the window known as slide interval W_s.

Stream inequality join. Let us assume two data streams $R = \{r_1, r_2, r_3, \ldots\}$ and stream $S = \{s_1, s_2, s_3, \ldots\}$; a stream inequality join can be defined as any inequality predicate $\bowtie_\theta$; $\theta \in \{<, >, \leq, \geq, \neq\}$ between a bounded set of tuples either from stream R and S ($R \bowtie_\theta S$). A tuple r_i from stream R compared with tuples from stream S using an inequality predicate. Additionally, it is inserted into its corresponding sliding window for future evaluations of new tuples from stream S (resp., $s_j \in S$).

Indexing and search with B^+ tree. Indexing data structures are employed to efficiently hold the sliding window contents for inequality join. The most common indexing data structure is B^+ tree [25]. It is a self-balancing data structure where data is stored in the leaf nodes of the tree; however, internal nodes act as indexes for search.

IE-Join. The IE-Join [13] data structure is designed for efficient inequality join of batch processing that comprises two phases: *initialization* and *probing*. The first step involves sorting the collected data, followed by constructing the permutation and offset array. Finally, a bit array is employed for query predicate evaluation. A *permutation array* is the position of tuple identifier of field b $r_{ib} \in R$ in the sorted array of field a $r_{ia} \in R$. Similarly, an *offset array* is the relative position of tuple $r_i \in R$ in the sorted array of the other relation $s_j \in S$ depending on the predicate operator.

2.2 Distributed Stream Processing System

DSPS uses a cluster of nodes for parallel processing of input data. It uses a client-server architecture where a new application in the form of a *DAG* is submitted to the server node [28].

Processing elements. DAG of streaming applications comprises many vertices v. Each vertex contains multiple processing elements (*PEs*). These *PEs* carry out the actual operation on streaming data, such as joining, filtering, aggregation, etc.

Stream partitioning. Stream data partitioning strategy is employed to downstream the tuples from vertex v_i to v_j of DSPS application, using partitioning strategy. The common methods of partitioning include hash partitioning, broadcasting a data unit to all downstream tasks, round-robin data partitioning, and direct mapping of the tuple from PE_i of vertex v_i to PE_j of vertex v_j [28].

Stream inequality join by DSPS. Indexing data structures are employed to hold the items of the sliding window by distributed *PEs* of *joiner* component [20]. A new tuple $\{r \in R, s \in S\}$, probes these data structures through distributed *PEs* [20]. Additionally, complex queries such as Q1 that involve more than

[4] https://github.com/AdeelAslamUnimore/StreamIEJoin/tree/master

one predicate require multiple vertices for evaluation. The *PEs* produce partial results that need further transformation for completeness.

2.3 Related Work

Real-time data processing requires retaining intermediate results in-memory rather than using I/O operations [1, 3–5, 9, 16, 22, 24, 31].

Sliding window-based stream join. Verwiebe et al. [30] provide a comprehensive review of different windowing strategies, however, many studies considered only the sliding window for stream join operation [7, 8, 14, 18, 19, 23, 25, 27, 34]. In handshake join [27], both streaming data flow in the opposite direction, where the predicate evaluation is only carried out during stream flow. Low latency handshake [23] (LLHJ) is an alternative solution for handshake join. Instead of queuing the tuple, LLHJ just expedites the windowing tuples to the next core to evaluate the predicate. Moreover, only one core or node is dedicated to holding a new incoming streaming tuple. Split join [19] is an extension of LLHJ, here the joining core or distributed node is divided into two parts such as the storage core or processing core. Similarly, chain index [18] and PIM tree [25] exploit the linked balanced tree data structure to produce query results specifically for non-equality-based streaming queries. Moreover, they use a coarse-grained tuple removal strategy from the sliding window. Our strategy uses two data structures, including mutable and read-efficient immutable components, to speed up the tuple evaluation process. Additionally, our proposed solution for stream inequality joins goes beyond the singular predicate.

Stream join in distributed stream processing system. Many studies target distributed stream join processing [5, 11, 12, 18, 26, 29, 32, 33, 35]. BiStream [18] is particularly designed to support window-based join, data aggregation, history based join. However, for non-equi join it employs chain indexes that raise issues of high insertion cost with larger chain length, AJoin [12] is a two-layered architecture with optimization and stream processing layers. The optimization layer periodically optimizes the execution plan, whereas the processing layer performs the stream join incrementally. FastJoin [35] handles load imbalance in DSPS with skewed input data. Our approach employs the round-robin strategy to distribute data to PO-Join instances to ensure load balancing

2.4 Challenges

We introduce a technique that combines both mutable (B$^+$tree) and immutable data structures (IE-Join) to perform inequality stream join for DSPS efficiently. However, we need to address nontrivial challenges that are associated with the tuple processing by the index tree data structure, merging from mutable index structure to immutable IE-Join, and distributed tuple processing by PO-Join instances

Index tree tuples processing. To process queries like Q1 and Q2, a naïve approach involves creating a hash table for the result set of both predicates. Then, a logical operation is required between these result sets. However, computing the hashed value for the result set tuples is expensive for these highly selective queries regarding memory and time. To address this issue, efficient mechanisms are required for predicate computation.

Merging from mutable to immutable design. The IE-Join [13] approach is a highly efficient method for *batch* inequality joins. However, it relies on sorted data items for permutation and offset array computation. Real-time sorting of streaming tuples is a challenging task. To tackle this challenge, we adopt a two-tier design comprising a mutable index tree and an immutable IE-Join data structure. The leaf nodes of index trees are utilized to sort streaming tuples. Additionally, the computation of permutation and offset arrays exhibit a quadratic time complexity, posing significant challenges for the real-time construction of the IE-Join data structure. Moreover, determining the appropriate merging threshold (δ) for transitioning between mutable to immutable data structures is a hard task. Although we use the slide interval as our merging threshold, this approach incurs extra overhead in computing the permutation array and partitioning it downstream to the immutable component, particularly for larger slide intervals of the window. In this case, an efficient mechanism for merging is required.

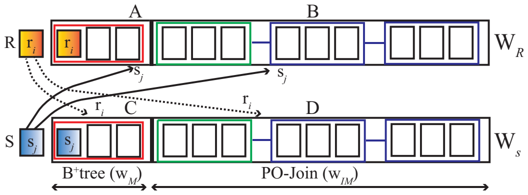


Figure 2: Proposed window-based SPO-Join

Challenges with distributed processing. In the mutable part of the SPO-Join, distributed components perform individual predicate operations. The partial results from these operations are then partitioned downstream to the distributed *PEs* of the logical operator. However, dealing with high insertion rates or different sizes of upstream indexing data structures can lead to correctness issues. To ensure correctness, an efficient data provenance strategy verifies the correct tuples from upstream predicate operators. This effective provenance is also required during the merge operation of mutable to immutable data structures.

For larger sliding window intervals, we divide the slide interval according to the number of downstream PO-Join *PEs*. This means that the contents of the sliding interval are held by several distributed PO-Join *PEs*. Consequently, it requires an efficient distributed state management approach for sliding window updates and removing the expired slide intervals.

3 PERMUTATION AND OFFSET-BASED DISTRIBUTED STREAM INEQUALITY JOIN

This section provides a detailed description of the proposed distributed stream inequality join (SPO-Join).

3.1 Solution Overview

The high-level overview of the proposed stream inequality join solution can be seen in Figure 2. Consider two streams, R and S, each building their sliding windows W_R and W_S, respectively. Each window is divided into two components: 1) an insert efficient *mutable component* (W_M) with streaming contents A for W_R and C for W_S, and 2) a search efficient *immutable component* (W_{IM}) with streaming contents B for W_R and D for W_S. The stream join between both streams is represented as $R \bowtie_\theta S$. This can also be further decomposed as $(A \cup B) \bowtie_\theta (C \cup D)$. In the proposed SPO-Join, a new tuple r_i or s_j from either stream must evaluate both components of the opposite stream for complete predicate evaluation.

Let us consider a use-case of tuple r_i, which belongs to the stream R. This tuple is inserted and indexed into W_M of W_R. Moreover, this tuple is broadcast to both components (mutable and immutable) of W_S for predicate evaluation $(r_i \bowtie_\theta C) \cup (r_i \bowtie_\theta D)$ (resp.,s_j). The merge operation is initiated in W_M of W_R or W_S at the threshold (δ), which depends on the time or count of tuples from the sliding window. Normally, for a smaller slide interval of window W_s, we use the slide interval as a merging threshold (δ). However, we subdivide the slide interval into sub-intervals for a larger sliding interval to avoid merging overhead, as detailed in Section 4. (We use the term $merge\_interval$ to refer to the sliding interval or sub-interval that is used for merging). In the merge operation, all tuples that are indexed in the data structure of W_M are merged into the linked set of search-efficient PO-Join structures, which is the immutable component W_{IM}. This operation frees up space for new tuples in W_M and reduces the indexing cost for new tuples in W_M. As analogous to the chain index [18], a coarse-grained tuple removal strategy is applied to the W_{IM} component of the sliding window, where the obsolete index from the linked set of PO-Join structure is removed for tuple removal threshold.

To process stream inequality join in a distributed and parallel manner, we use the system model illustrated in Figure 3. For complex queries like Q1, which involve two opposite streams R and S and their intersection, we divide the execution into three components: mutable (B$^+$tree and offset computation), permutation computation, and immutable (PO-join) structure. For the mutable design, we have three operators that perform operations in a pipeline-parallelism manner. Two parallel operators evaluate individual predicates and then transform their partial results to the logical operator using hash partitioning. This operator contains multiple parallel processing elements on several distributed nodes that can simultaneously perform logical operations on upstream data. Similarly, operator PO-Join evaluates the W_{IM} tuples. It contains multiple processing elements, each of which holds the linked list of the PO-Join data structure. The step-by-step SPO-Join procedure is explained by the Algorithm 1.

Algorithm 1: Distributed stream inequality join (SPO-Join)

```
Input: Tuple $r \in R$ (resp., $s \in S$), Window lengths ($W_L$) and slide interval
    ($W_s$), Processing elements ($PEs$)
Output: Stream join result for each input tuple ($r$ or $R$)
Broadcast $r$ from router to $W_M$ ($PE_1$ and $PE_2$) and $W_{IM}$ ($PEs_{PO-Join}$) of
    joiner (resp., $s \in S$) as depicted by Figure 3;
Perform inequality join between $r$ and index structures of stream $S$ in $W_M$
    (explained by Figure 4), as well as with $W_{IM}$ of PO-Join structures
    (detailed by Algorithm 4 and Figure 5) (resp., $s \in S$);
Insert $r$ into the index structure of stream $R$ in $W_M$ as depicted by
    Figure 3 (resp., $s \in S$). ;
Update the $merge\_interval$ counter. ;
if $merge\_interval > \delta$ then
    Compute offset array $O_{r,s}$ in $PEs$ of $W_M$ (Algorithm 3);
    Partition $O_{r,s}$ to downstream PEs of $W_{IM}$ ($PEs_{PO-Join}$) in a
        round-robin fashion;
    Compute permutation array ($P$) from index structures of $W_M$ to the
        dedicated $PEs$ (Algorithm 2);
    Downstream permutation array ($P$) to $PE_{sPO-Join}$ of $W_{IM}$ round
        robinly;
    Insert $O_{r,s}$ and $P$ into the linked list of PO-Join $PE$;
    Update the state of $W_{IM}$ among $PEs_{PO-Join}$;    // Check slide
        interval $W_s$ of window threshold $W_L$
    Re-initialize the $merge\_interval$
```

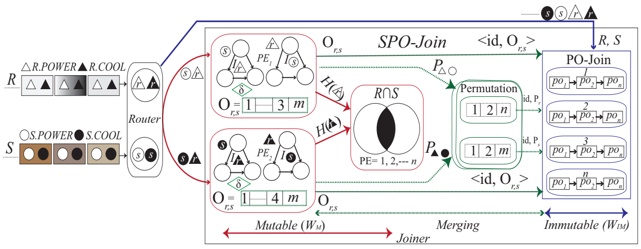


Figure 3: SPO-Join model (Q 1)

3.2 Mutable Component

A new tuple from streams R or S are indexed into the mutable components. These tuples are also probed against the underlying index structures based on predicate as shown by Figure 3. Mutable components contain the subset of the sliding window W_L tuples.

Tuple insertion. We utilize our first use case Q1 and Figure 3 to understand the insertion process. A new tuple is inserted into the *router* part of the stream join model. Let us consider a tuple from stream R, $r = \{id, [R.POWER, R.COOL]\}$. Initially, the *router* split and partition this tuple to both *PEs* of mutable components for predicate evaluation. Additionally, it assigns a time unit to each tuple based on its arrival order in the router component [18]. This approach helps to reduce conflict between tuples whose event timestamps are identical, especially for the time-based sliding windows. The tuple from stream R with $r = \{id, R.POWER\}$ is partitioned to PE_1 and $r = \{id, R.COOL\}$ to PE_2. Furthermore, this tuple is indexed into I_r and probed against I_s in PE_1 (resp., PE_2).

Tuple evaluation. Akin to insertion, searching is also done on the indexing data structures. Let us continue the use-case Q 1 with Figure 4. For tuple $r_i < id : 202, R.POWER : 3000 >$, the identified leaf node (N) of index structure I_s contain $S.POWER$=3300 as shown by Figure 4 with ①. However, the result set includes all tuples in the leaf nodes where $r_i < S.POWER$. Instead of a hash table, we introduce a bit array with the same length of mutable data structure as depicted by Figure 4 with ②. The identifiers of the mutable window tuples act as index positions for the bit array. In this case, all bit positions of the bit array are set to true where the new tuple r_i satisfies the predicate condition. After predicate evaluation, this bit array is partitioned towards the logical operator PE using hash partitioning (H_1(id: 202)). Similarly, for r_i : (id:202, $R.COOL$:500), the predicate evaluates r_i> $S.COOL$ and creates a bit array. This bit array is forwarded to logical operator PE using hash partitioning and performs a logical operation as depicted by Figure 3 with ③. This approach may introduce processing guarantee issues. However, we mitigate these concerns by employing hash partitioning from the predicate operator PEs to the logical operator PEs, along with a small hash table in the PEs of the logical operator. This hash table stores the partial results of upstream PEs, ensuring correctness for the bitset logical operator as elaborated in Section 4.3. Once both bit arrays arrive at the PE of the logical operator, an *AND* operation is initiated between the bit arrays to obtain the complete query result.

Time complexity. The cost of inserting a tuple t_i into the indexing data structure is $O(\log n)$. For range searches, the cost is $O(\log n + m)$, where m is the cost of traversing from the identified node N to all nodes that meet the specified predicate condition. A single bit flip in the bit array is $O(1)$ but with m bits in the array it can reach $O(m)$ in the worst case. Additionally, $O(m)$ is the computation cost of logical operation on bit arrays.

Memory cost analysis. Let $\mathcal{I}_M$ be the cost of memory consumption by all indexing data structures for mutable components. Then, $\mathcal{I}_M$ can be represented by the following Equation 1:

$$\mathcal{I}_M = \sum_{j=1}^{n} \mathcal{i}_M + \mathcal{c} \quad (1)$$

In Equation 1, n shows the number of indexing structures for the fields. Here $\mathcal{i}_M$ is the memory cost of a single field, while $\mathcal{c}$ denotes the price of buffering for logical operations and data provenance. The cost of $\mathcal{c}$ is much smaller than the cost of $\mathcal{i}_M$ ($\mathcal{c} << \mathcal{i}_M$).

3.3 Merging to Immutable Component

In the SPO-Join, we merge the mutable window (W_M) into an immutable data structure, which depends on the merging threshold (δ) as depicted by Figure 3 with conditional box. Initially, we use the slide interval W_s as δ, either on a count-based or time-based stream sliding window ($\delta = W_s$). However, this method incurs additional merging cost for slide intervals of larger size. To address this challenge, we propose subdividing the slide interval into sub-intervals, where each sub-interval acts as δ based on the number of downstream processing instances of the PO-Join component ($\delta = W_s/|PEs_{PO-Join}|$). This mitigates overhead associated with larger slide interval merging, however, introduces state synchronization overhead, as detailed in Section 4.2.

The leaf nodes of the B$^+$tree indexing structure in mutable components contain sorted data items. These nodes also include the explicit addresses of the predecessor and successor nodes, making it less expensive to scan sorted data items to compute permutation and offset array for the PO-Join structure.

The offset array is computed on *PEs* of mutable component and then partitioned to the PO-Join operator *PE* with an identifier and an offset value (<id, O$_{r,s}$>) as shown by Figure 3. However, the permutation array is computed between the intermediate *PEs* among mutable and immutable components of SPO-Join as depicted by Figure 3. Initially, the permutation array is computed between fields (*R.POWER*, *R.COOL*) of the same streaming data such as *R* (resp., *S*) and then forwarded toward PO-Join operator *PE* with an identifier and a permutation value <id,P$_r$> (resp., <id,P$_s$>).

Permutation computation. It involves generating an array that indicates the position of tuples with different fields of the same stream w.r.t the tuple identifier. For instance, in query Q1 and Figure 5, consider the tuple from stream *R* with field *R.COOL* and identifier r_2 (the first tuple in a sorted order). The position of r_2 in the sorted array of field *R.POWER* is 4. This array aids in reducing the cost associated with intersection operations [13].

Algorithm 2: Permutation computation of stream *R*

```
Input: Sorted $R_a[Tuples]$, Sorted $R_b[Tuples]$
Output: Permutation-Array (P)[]
1 counter ← 1
2 tmp[] ← 0 ;                       // Initialize an empty array
3 for $ID_i \in R_a$ do
4     tmp[ID$_i$] ← counter ;   // ID is an identifier of the tuple
          which is assigned by the router component of SPO-Join
5     counter ← counter + 1
6 for $ID_j \in R_b$ do
7     P[j] ← tmp[ID$_j$] ;   // ID remains the same among fields of
          tuple
8 tmp[] ← 0
```

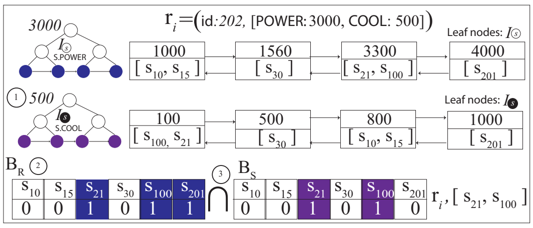


Figure 4: Mutable-part stream join (Example Q 1)

The process of computing permutation array is outlined in Algorithm 2. The algorithm takes a sorted array of tuples from both fields *a* and *b* for the identical streaming data *R* as input and produces an output that provides the *position* of field *b* in the field *a* (*a* and *b* are field data as *R.POWER* and *R.COOL* in Q1). Initially, the algorithm initializes a temporary counter on line 1 and a temporary array that holds the identifiers of tuples on line 2. From line 3 to line 5, the algorithm iterates through all tuples of field *a* and fills the temporary array with an incremental counter. During this process, the field identifier of *a* is considered an index location in the temporary array. The algorithm then iterates through the opposite field items *b*, on line 6. Finally, the permutation array is filled with the temporary array where field *b* identifier acts as index position for the temporary array as depicted by line 7 and then partitioned toward the PO-Join operator *PE*.

Offset computation. Offset computation requires identifying the relative location of tuples from a sorted array of different fields of opposite streams (*R* and *S*) depending on the predicate. For example in Q1 and Figure 5 the predicate exists between *R.COOL* and *S.COOL*, the relative location of the first sorted tuple from *R.COOL* say r_2 = [1600] in *S.COOL* should be 2. This array helps to identify the location of the opposite streaming tuple that satisfies the predicate condition in a constant time, further speeding up the bit array scanning operation [13].

Algorithm 3 outlines the process for computing offset, where the input is the indexing data structures for the opposite stream I_R and I_S of *R.COOL* and *S.COOL* (resp., *R.POWER* and *S.POWER*), and the output is the relative location of data items of I_R in I_S.

Algorithm 3: Offset computation between opposite fields of predicate relations

```
Input: Index-tree($I_R$), Index-tree($I_S$)
Output: Offset-array [$r_i, s_i$] ($O_i$)
Offset-index ← 0
for $k_r \in I_R$ do
    if Offset-index==0 then
        Offset-index ← $I_S$.search($k_r$)
        Offset-array [$k_r$, Offset-index]
        continue;
    else
        for $k_s$[Offset-index] ∈ $I_S$ to $k_s.Length$ do
            if $k_s \geq k_r$ then
                Offset-index ← $k_s.pos$ ;      // offset index is
                    updated with new position
                Offset-array [$k_r$, Offset-index]
                break;
            else
                Offset-array [$k_r$, $k_s.Length + 1$]
```

To begin with, the offset index position is initialized to 0 in line 1. Starting from line 2, all keys of I_R are iterated linearly. If the offset index position is 0, the algorithm begins searching for the key k_r in the opposite index data structure I_S from the start, updates the index position with the newly identified position of k_r in I_S in line 4, and sets the offset location of k_r to the updated relative index position in line 5. The outer loop then continues to the next key in I_R. Similarly, if the offset index position is greater than 0, the algorithm scans the keys of I_S from the newly offset position to the end, as depicted by lines 8 to 12. If the key $k_s \in I_S$ is greater than or equal to the key $k_r \in I_R$, then the offset index position is updated with the position of k_s in I_S as explained by line 11. If k_r does not find any resemblance, then the offset position for k_r is set to the number of keys plus one, as highlighted in line 14 of Algorithm 3.

3.4 Immutable part Tuple Evaluation

Algorithm 4 explains the probing phase in the immutable part (PO-Join) of the proposed stream inequality join. The input to the algorithm is a new tuple t_i, the total linked PO-Join data structure, and the number of cores available in the computing node.

In line 1, we initialize the number of threads equal to the number of cores available in the node. However, another choice can be to initialize these threads with the size of the linked list. This process could increase the overhead of more context switching between threads when the available cores are limited specifically for commodity hardware. Lines 3 to 14 describe the process of the PO-Join data structure. However, we use a lock to ensure that multiple threads will not get the same linked list index. In line 5, we check the size of the linked list and ensure that all linked list elements are assigned to threads. Each thread computes the join on a new tuple t_i with a selected linked list index of PO-Join, as depicted by line 11. Similarly, line 12 indicates the updating of the sliding window counter.

Tuple evaluation with PO-join. Figure 5 depicts the process of stream inequality join by the immutable component of SPO-Join. Figure 5 with ① represents the required offset and permutation arrays from the index trees.

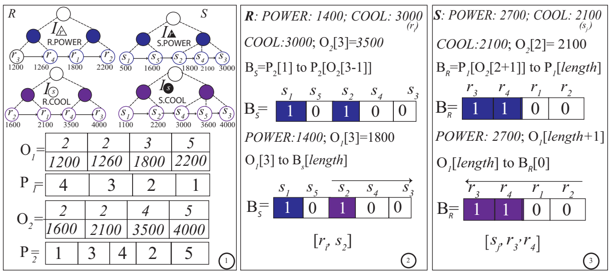

R
S
I_R
R.POWER
I_R
S.POWER
r_3 r_4 r_1 r_2
1200 1260 1800 2200
s_1 s_5 s_2 s_4 s_3
500 1600 1800 2100 3000
I_S
R.COOL
I_S
S.COOL
r_2 r_1 r_4 r_3
1600 2100 3500 4000
s_1 s_2 s_4 s_5 s_3
1100 2200 3000 3600 4000
O_1= 2 2 3 5 / 1200 1260 1800 2200
P_1= 4 3 2 1
O_2= 2 2 4 5 / 1600 2100 3500 4000
P_2= 1 3 4 2 5
①
R: POWER: 1400; COOL: 3000 (r_i)
COOL:3000; O_2[3]=3500
B_S=P_2[1] to P_2[O_2[3-1]]
B_S= 1 0 1 0 0
POWER:1400; O_1[3]=1800
O_1[3] to B_s[length]
B_S= 1 0 1 0 0
[r_i, s_2]
②
S: POWER: 2700; COOL: 2100 (s_j)
COOL:2100; O_2[2]= 2100
B_R=P_1[O_2[2+1]] to P_1[length]
B_R= 1 1 0 0
POWER: 2700; O_1[length+1]
O_1[length] to B_R[0]
B_R= 1 1 0 0
[s_j, r_3, r_4]
③

Figure 5: Two-way join (Example: Q1)

In a stream join, the tuple can come from both stream R or stream S. Therefore, we have provided descriptions for both tuples against Q1 along with an example. 1) Initialize an empty bit array (B_S) for stream S upon receiving a new tuple $<r_i, R.POWER : 1400; R.COOL : 3000>$ from stream R. 2) To find the relative location of a tuple in an offset array, perform a binary search on the offset array O_2 for the field *R.COOL*: 3000. When performing the binary search, the position 3, which corresponds to 3500, is found. Since this value is greater than the cooling value of 3000, use position 2 of offset array O_2 such as position found minus one. 3) Set bit positions of B_S to 1 for the permutation array $P2[1]$ to $P_2[O_2[2]]$. In this example, locations $P_2[1]$=1 and $P_2[2]$=3; now bit positions 1 and 3 should be 1 of B_S. 4) Perform a binary search for the field *R.POWER* tuple in the offset array O_1. 5) Start scanning the B_S from $O_1[position + 1]$ to the end of offset array O_1. For example, if we perform a binary search for $R.POWER$=1400 and it returns 1800, we should consider position 3 as the starting point (1800 is already greater than 1400) and scan the bit array from $O_1[3]$ to the end of B_S. All identifiers of the identified 1's in the B_S are result set. In this case, the result set for r_i tuple is s_2 as shown by Figure 5 with ②. To efficiently evaluate the new tuple from stream S say $<s_i, POWER : 2700; COOL : 2100>$, we perform a binary search on the existing O_2 for *S.COOL* and set 1 in the bit set B_R from $P_1[O_2[position]]$ to the end of P_1. Moreover, we also identify the position of *S.POWER* in O_1 and start scanning the bit array from the identified *position* of O_1 to the start of B_R. All identified 1's locations of the bit array are the result set as shown by Figure 5 with ③. Updating the bit arrays and scanning the tuples can be changed based on the predicate value as explained in [13] .

Tuple evaluation during merge operation. When a merging interval (δ) is received in the mutable part of the sliding window as shown by Algorithm 1 line 5, a flag tuple is sent to the chosen downstream *PE* of the PO-Join operator to initiate the merge operation. Once the downstream *PE* receives the flag tuple, an empty queue is dedicated to start inserting incoming tuples from the source. After the merge operation completes and builds the respective PO-Join structure in that *PE*, the tuples from the queue start probing the newly merged PO-Join structure until the queue becomes empty.

Tuple removal. Akin to the chained index [18], tuples are removed in coarse granular way for both small and large slide intervals from W_{IM}. For smaller slide intervals ($\delta = W_s$), the expired sub-interval is removed from a single downstream *PE* of the PO-Join operator. However, for larger slide intervals where we subdivide the slide interval ($\delta = W_s / |PEs_{PO-Join}|$) contents to all downstream $PEs_{PO-Join}$. In this case, all *PEs* of the PO-Join operator hold the expired tuples of the sliding window. The old indexes of PO-Join linked list on each *PE* that contain the expired slide interval contents are removed on each expiration. We provide detailed information on tuples state management in Section 4.2.

Time complexity. Assigning an index value to the temporary array requires $O(n)$ time. Similarly, it takes $O(n)$ time to obtain the permutation location of the field b ($R.COOL$) in the

Algorithm 4: PO-Join: Tuple t_i evaluation on PE$_i$ of W_{IM}

```
   Input: $t_i$; LinkedList⟨PO-Join$_{1 \to n}$⟩; |cores|
   Output: $t_r \bowtie_\theta S_{IM}$ (join result)
1  Initialize-threads ⟵ |cores|
2  Current-index ⟵ 0
3  while true do
4      lock ;  // Ensure multiple threads do not access the same
         [PO-Join]
5      if Current-index < LinkedList.size() then
6          PO-Join ⟵ LinkedList.get(Current-index)
7          Current-index ⟵ Current-index + 1
8      else
9          break ;              // All elements have been accessed
10     unlock
11     $t_i \bowtie_\theta$ PO-Join ;                  // Explained with Figure 5
12 Update the state of $W_{IM}$ for $PE_i$
```

field *a* (*R.POWER*) (resp., *S.COOL* and *S.POWER*). Here, *n* is the total tuples from stream *R*. The total time complexity for the permutation array of stream *R* is $O(n + n)$.

The first key from I_R requires $O(\log m)$ to find its location in the leaf node of I_S that serves as a reference point. All other keys from I_R start searching the relative location from that reference point. Additionally, this reference location is updated for each r_i. So, the total complexity can be written as $O(n + m)$. We require two binary searches on sorted data items for the probing phase. The time complexity for searching a tuple is $O(\log n)$. Moreover, filling the bit array for the permutation location has constant time $O(1)$.

Memory cost analysis. Equation 2 represents the memory cost of SPO-Join's immutable component, denoted by $\mathcal{I}_{IM}$.

$$\mathcal{I}_{\mathcal{IM}} = \sum_{j=1}^{m} x(P_i + O_i) + \beta \tag{2}$$

In Equation 2, the *m* represents the total number of PO-Join *PEs* of the streaming window that exists on distributed *PEs*. The *x* represents the number of predicate relations θ. P_i and O_i represent the cost of permutation and offset array for the opposite stream of relations. Similarly, β represents the cost of a bit array for an input tuple *r* or *s* that is further used for finding the matching tuples.

4 DISTRIBUTED JOIN PROCESSING

In this section, we discuss different aspects of efficient distributed query processing for SPO-Join.

4.1 Data Partitioning

When dealing with the mutable component, the results of the predicate operators are hash partitioned to the logical operator *PEs*. Hash partitioning ensures correctness as detailed in Section 4.3. The offset array is partitioned to the downstream instances in a round-robin way, however, the permutation array requires intermediate *PEs*. First, the tuples are downstream from the mutable operator's *PEs* to the intermediate dedicated *PEs* of the permutation array using direct partitioning. From the dedicated *PEs*, the data is partitioned using a round-robin scheme towards PO-Join *PEs* to equally utilize the instances of PO-Join operators.

4.2 State Management

In SPO-Join, we employ multiple *PEs* for immutable components. We use three different strategies to manage the state of the sliding window among these distributed *PEs*, particularly for count-based (as time-based is straightforward).

Firstly, the selection of downstream *PEs* depends on the ratio of W_L to W_s on available nodes (a single node may contain more than one *PE*) for smaller-size slide intervals. The permutation and offset arrays from the mutable component are distributed in a round-robin style. When a new permutation array is assigned to the downstream *PE*, it signals that the existing sliding interval W_s on this *PE* has expired and should be removed from its data structure.

Secondly, for larger sliding intervals, the approach described above increases the cost of merging. To solve this problem, we propose dividing the sliding interval W_s with total instances of downstream processing elements ($PEs_{\text{PO-Join}}$). In this case, when we send the permutation array to PO-Join *PE*, we also send the size of merging tuples to all other *PEs*. This is depicted in Figure 6 (A-left). We check the tuple removal condition for new tuples and remove the first index of the linked PO-Join structure from PEs.

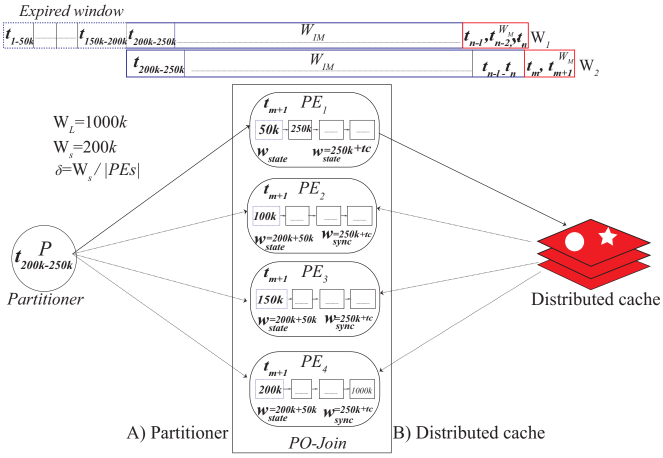


Figure 6: State-management for sliding window

Thirdly, to simplify the communication between *PEs*, we have adopted a new strategy that uses a distributed cache as depicted by Figure 6 (B-right). Instead of sending the sliding count to other *PEs*, the first *PE* continuously sends its window state to the distributed cache after a regular interval. Similarly, the other *PEs* synchronize their local windows from the cache after a specific interval of time or count. For every new tuple, the window updating procedure is used to remove the first index of the linked PO-Join structure from the *PE*.

Example Let us consider an example of a sliding window with a window length (W_L) of 1000K, a slide interval (W_s) of 200K, and 4 downstream *PEs* of the PO-Join operator. The slide interval δ is divided into sub-intervals based on the downstream $PEs_{PO-Join}$, as shown in Figure 6. Each sub-interval from upstream processing instances is downstream to PO-Join *PEs* in a round-robin way. Tuples of W_s exist on all *PEs* of PO-Join. For strategy A), tuples from 200K to 250K are partitioned to PE_1. However, the count of these tuples (50K) updates the local window state of all other processing elements. Figure 6 illustrates this state management for the sliding window. For strategy B), each evaluating tuple t_{m+1} also updates the local window state of PE_1 as $w_{state} = 250 + t_c$. Here t_c depicts the local tuple counter for PE_1. This update is then propagated to the distributed in-memory cache, and all other processing elements synchronize their local window state from this cache (DC).

False positives. These schemes may produce a false positive for some tuples. To better understand this, let us take an example with the help of Figure 6. Here, we have two sliding windows W_1 and W_2, where W_2 starts after the slide interval of W_1 (200K) tuples. The new tuples of W_2, such as t_m and t_{m+1} are broadcasted to both W_M and W_{IM}, where the immutable component *PEs* produce join results faster than mutable windows. However, the local state of the window among PO-Join *PEs* is updated depending on each merge operation from the mutable part. So, the result set for t_m or t_{m+1} of W_2 may perform join on expired sub-intervals of W_1 that exist on distributed processing instances due to delay in removing expired tuples, especially for high input data rate. The distributed cache-based approach is employed to reduce the rate of these false positives. The first processing element, PE_1, updates the window state depending on each evaluating tuple, i.e., t_m or t_{m+1}, and partitioner information. Others *PEs* sync their window state to PE_1, reducing false positives for new tuples, though it may still introduce expired tuple results for t_{m+1}.

4.3 Processing Guarantees

Mutable. We use hash partitioning to ensure that tuples with the same *id* are processed by the same processing element from the pool of *PEs*. However, hash partitioning may sometimes result in incorrect outputs if different keys are processed by the same processing element, especially when the logical operator *PEs* is limited with a high insertion rate of tuples. Let us consider a new tuple t_i from stream *R*, comprising $I_{R.COOL}$ and $I_{R.POWER}$ for the query Q1. The result set for $I_{R.COOL}$ is produced more quickly than for $I_{R.POWER}$ due to a less dense index structure or low selectivity. The outputs from both index structures are then sent downstream to PE_m of the logical operator using hash partitioning $H(i)$ as depicted by Figure 3. As a result, the output from $I_{R.COOL}$ arrives earlier than the output from $I_{R.POWER}$, forcing it to wait (out-of-order arrival at PEs of the logical operator). Meanwhile, another tuple t_{i+1} generates the result for $I_{R.COOL}$ and through hash partitioning, it selects the same downstream PE_m which overrides the results of t_i from $I_{R.COOL}$. Subsequently, the logical operation is performed between the results of t_i of *R.POWER* and t_{i+1} of *R.COOL* on PE_m. To overcome this issue, we use a lightweight hash table. The hash table first checks if the key is already present in its data structure then logical operation can be performed among bit arrays for final result accumulation, finally, the key is removed from the hash table.

Immutable. During the merging of tuples from the mutable part and computing the permutation and offset array, there may be data integrity issues, especially for high-input data tuples or different sliding intervals between streams. To solve this problem, each batch of the mutable component (permutation or offset array) is assigned a separate identifier *id*. In the PO-Join operator, a smaller hash table is maintained that holds these batches along with their respective *id*. For Q1, each record in the hash table is represented as $id_i :< PE_1, PE_2, O_1, O_2 >$. The PO-Join structure is created and inserted into the linked list only from this hash table. After the insertion of these batches into the linked list, the *id*, along with all of its payload, is removed from the hash table.

5 EVALUATION METRICS AND RESULTS

In this section, we describe the evaluation metrics, dataset, experimental setup, and discussion of the results for distributed stream inequality join algorithms.

5.1 Metrics

We choose these evaluation metrics from the benchmark on DSPS [11] and other related studies on stream processing [18, 25].

Throughput. It is the number of tuples that join algorithms process per second, reflecting their data processing capacity.

Latency. For this study, we have chosen to analyze two types of latency: *event time latency* and *processing latency* [11]. Here the event time latency refers to the time taken for a tuple to be completely processed, from the moment it enters into the router component of DSPS until it is processed by our SPO-Join components. This type of latency also includes any network-related costs associated with processing the tuple. On the other hand, processing latency refers to the time taken for a tuple to be processed from the moment it enters the *joiner* until it is finally processed.

Match rate [25]. The sliding window has a dynamic matching rate for every new tuple that satisfies the query predicate conditions. We synthetically varied the matching rate for each sliding window and observed the SPO-Join performance.

Scalability. In the context of streaming, scalability can be defined using two terms: vertical scalability (scale up-increases the number of processing instances) and horizontal scalability (scale out-increases the number of machines) [25].

Correctness and completeness. For the mutable part of SPO-Join, we assess the correctness of tuple processing, wherein a singular tuple undergoes complete processing by diverse *PEs*. Similarly, for the immutable part, we evaluate proposed state management techniques on the *false positive* with varying insertion rates.

We conduct experiments by varying parameters such as window length, slide intervals, *PEs*, number of machines, matching rate, and merging threshold, depending on each join algorithm.

5.2 Datasets and Queries

We used two real-time datasets and one syntactic dataset, along with three queries for our experiments as summarized in Table 1.

- **Real datasets**. We use the New York taxi dataset that was introduced in the DEBS Great Challenge of 2015 [10]. It consists of geospatial data for an online taxi service operating within the city. This data includes information about each taxi trip, such as the start and end times, the location, and the distance traveled. It contains 172M tuples. Additionally, we utilize the BLOND [15] dataset, which contains data on the current and voltage readings of electric appliances in a building. We used a subset of the BLOND-250 dataset that consists of 50 days of readings taken on every hour. However, we use 2B tuples for our evaluation.
- **Synthetic dataset**. We use a synthetic dataset with varying match rates created by the DSPS engine at runtime. The total size of the synthetic dataset is 32M tuples.

Q 3: Analysis of NYC Taxi Trips: Distance vs. Fare

```
SELECT trip.ID FROM NYC
 WHERE NYC.trip_dist1 > NYC.trip_dist2
 AND NYC.trip_fare1 < NYC.trip_fare2
WINDOW AS ( SLIDE INTERVAL 'D'  ON 'W');
```

To evaluate the proposed stream inequality join algorithm, we have conducted experiments using self-join (Q 3), band join (Q 2), and cross join (two-way join) (Q 1) queries. We employ the New York taxi dataset (NYC) for self-join and band join. Additionally, we use the BLOND dataset for cross-join query Q1. This involved calculating the power for data center racks and cooling infrastructure using various files of current (I) and voltage (V) readings, with their product used as power. Moreover, we opt count-based window for Q1 and Q3 and time-based for

Table 1: Inequality queries type and datasets description

Query	Dataset	Tuples	δ ranges	Join type	Bandwidth
Q 3	NYC-taxi [10] Synthesize	172M 32M	10K-100K	Self join	
Q 2	NYC-taxi	172M	1min-5min	Band join	$3x10^{-2}$
Q 1	BLOND [15] Synthesize	2B 32M	20K-300K	Cross join	

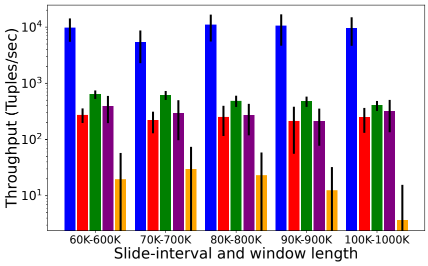


Figure 7: Throughput for self join Q3

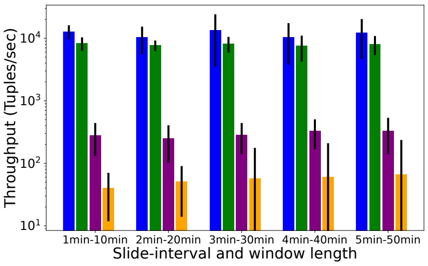


Figure 8: Throughput for band join Q2

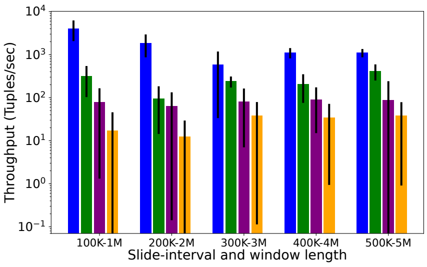


Figure 9: Throughput for cross join Q1

Q2 with bandwidth of $3x10^{-2}$. The same queries are adopted for synthesized data.

5.3 Experimental Setup

We use 10 machines with varying RAM sizes, ranging from 4GB to 12GB, and disk capacities ranging from 120 GB to 500 GB. All machines are connected to the same network. We have utilized Apache Storm 2.4.0 as a benchmark DSPS, implementing all join algorithms with the same semantics of Apache Storm, such as *Spout* and *Bolt* to maintain system uniformity. We do not employ other streaming systems, such as Flink and Spark Streaming as the former lacks built-in support for theta join [2], while the latter only supports mini-batches. *Nimbus* is used as a master service, whereas *Supervisor* as a workers service. *Zookeeper* maintains coordination among nodes. Nodes are synced through the local NTP server on *Nimbus* node. Apache Kafka 3.5.0 and *Redis* are employed for streaming the data to the DSPS and as an in-memory database for distributed cache. Moreover, we employ at least once processing guarantee to ensure complete reliability against message loss.

5.4 Results and Discussion

In this sub-section, we present experimental results for stream inequality join algorithms.

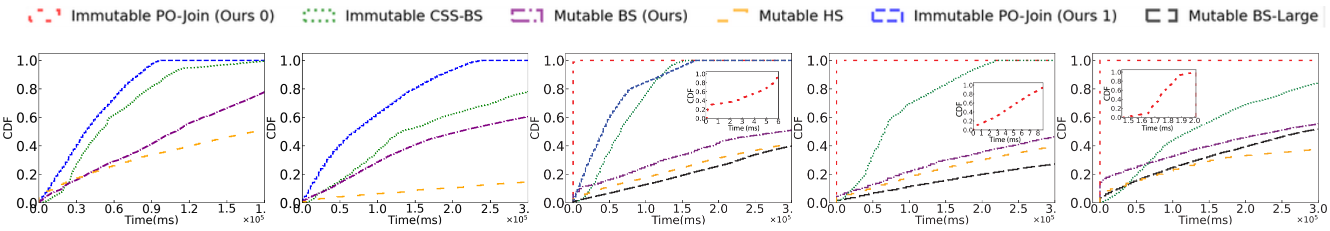
Mutable-Immutable join structure. Figure 7 to Figure 10 provide results for distinct mutable and immutable design-based stream join strategies against real-world datasets. Figure 7 represents the throughput for Q3 using sliding intervals ranging from 60K to 100K and window sizes from 600K to 1M tuples. The *PEs* for the immutable part also vary from 6 to 10. The y-axis depicts the mean and standard deviation of tuple processing throughput. The results indicate that the immutable PO-Join component outperforms the hash-based and bit-based immutable CSS-join algorithms by an average of 35x and 16x, 25x and 12x, 43x and 22x, and 57x and 31x, respectively, as the window size increases. Moreover, for the mutable component, the graph shows that the bit-based stream join performs 19x better than the hash-based mutable component for the 60K slide interval. However, when comparing the lower limit of the standard deviation for the bit-based join and the upper limit of the hash-based join algorithm, the proposed bit array-based mutable join is 3.6x times better than the alternative. Additionally, the mutable part of the SPO-Join algorithm shows superior performance than the hash-based intersection join, with a mean of 9x, 12x, 17x, and 44x as the sliding window size increases from 70K to 100K.

Figure 8 shows the average throughput and standard deviation for Q2 for an increasing time-based sliding window. The figures indicate that the immutable part of the proposed SPO-Join has an average tuple processing throughput that is 1.5x, 1.3x, 1.6x, 1.3x, and 1.5x better than the CSS-based data structure. Similarly, the bit-based join strategy has achieved 7x, 4.9x, 5x, 5.5x, and 4.9x better throughput for the mutable component than the hash-based join mutable join strategy. Moreover, the difference in standard deviation is 1.6x, 3.8x, 4.8x, 2.0x, and 3.02x for immutable components and 5.3x, 3.9x, 1.02x, 1.09x, and 1.14x for mutable join strategies.

Figure 9 depicts the throughput of two-way join query Q1 for mutable and immutable data structures. In this case, the *PEs* are fixed to 10 for both immutable data structures. For 100K and 200K slide intervals, the proposed PO-Join has 12x and 19x better throughput than the alternative immutable structure. However, the mutable designs depict that bit-based join has 4.6x and 5.2x superior performance than the hash-based alternative join strategy. Similarly, for a 300K slide interval with 3M tuples of window length, we divide the slide interval into the available processing elements to minimize the merging cost. The result indicates that the proposed PO-Join structure has 2x better performance than the CSS tree-based join strategy. Moreover, our bit-based mutable component also performs 2x times better than hash-based join. Analogous performance improvement is noted for 400K-4M and 500k-5M slide interval and window length where PO-Join has 5.38x and 2.6x superior performance than immutable CSS join structure. Similarly, bit-based mutable join has 2.6x and 2.23x time better performance than hash-based join.

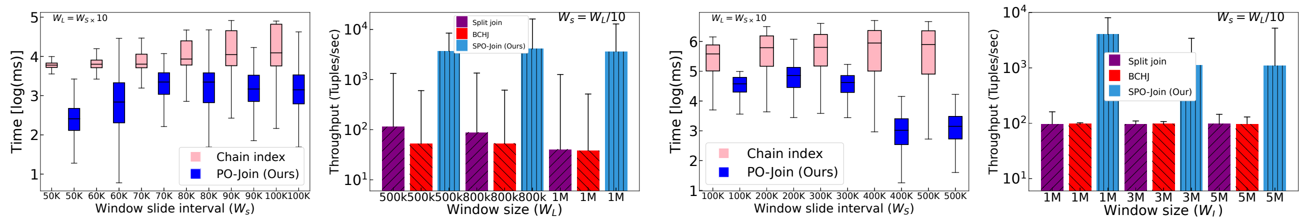
Figures 10a to 10b display the commutative frequency distribution (CDF) of event time latency for Q1. In Figure 10a, we observe the results for a 100K sliding interval with a 1M window size. The graph shows that the immutable part of the stream PO-Join takes 37 seconds for the 50th percentile of tuples, while the CSS-tree-based immutable part takes 49 seconds. The mutable component spends 109 seconds for bit-based join and 173 seconds for hash-based join. Similarly, for the 75th percentile, the proposed stream PO-join takes 62 seconds while the alternative consumes 80 seconds. Moreover, the proposed mutable part exhibits 2x better performance than the hash-based competing solution. The 95th percentile for the immutable part of the PO-Join algorithm has 1.5x superior performance than the CSS tree-based immutable part. Additionally, the mutable part is twofold better than the alternative. Similarly, Figure 10b depicts analogous performance improvement for SPO-Join.

Figure 10c shows the CDF plot for 300K and 3M window size. In this case, we employ two approaches for merging threshold δ selection: (1) $\delta_1 = W_s$, (2) $\delta_2 = W_s / |PEs_{PO-Join}|$. Results depict that for the 50th percentile; the divided slide interval-based (δ_2) PO-Join structure is 70x better than the normal slide interval (δ_1). Moreover, for the mutable part, the divided sliding window with



(a) Latency 100K-1M (b) Latency 200K-2M (c) Latency 300K-3M (d) Latency 400K-4M (e) Latency 500K-5M

Figure 10: Event time latency for Q1 for BLOND [15]



(a) Latency: CI and PO-Join for Q3 (b) Throughput: NLJ & SPO for Q3 (c) Latency: CI and PO-Join for Q1 (d) Throughput: NLJ & SPO for Q1

Figure 11: Chain index and nested loop strategies comparison with stream SPO-Join

B$^+$tree has 1.5x time better in join computation than the complete slide interval. For the 75th percentile, the divided slide interval has 44.6x and 1.7x for immutable and mutable parts superior to its counterpart (full slide interval). Additionally, Figure 10d and Figure 10e adopt slide interval division, where the results show superior behavior of SPO-Join.

Both designs offer two components for the whole stream inequality join structure. The majority of tuples from sliding windows are retained by the immutable component of the data structure. However, a small portion of tuples are held and executed by the mutable component. A new streaming tuple employs binary search for PO-Join data structures, which then scans a consecutive memory location of sorted tuples. However, in the CSS tree, tuples exist on the tree's leaf nodes, and these tuples are packed using blocks where distinct nodes are linked together. The scan on linked nodes is more expensive than searching the consecutive memory locations. Additionally, the intersection procedure by PO-Join is less expensive as it contains the memory location of tuples that have a constant cost to fill or scan the bit array. The CSS-based join structure takes more time to process a tuple and requires an extra level of data provenance for correctness.

For larger windows, the sliding window interval is divided into sub-windows and distributed these windows to the downstream instances in a round-robin way with an efficient state-management strategy. This process reduces the merging cost of tuples from mutable to immutable components. Furthermore, it enhances the processing capacity of tuples, subsequently reducing the waiting time of tuples in the queue.

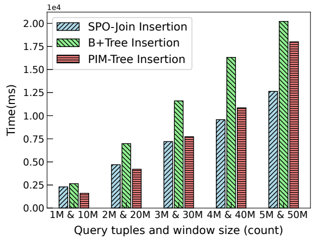


Figure 12: Insertion cost

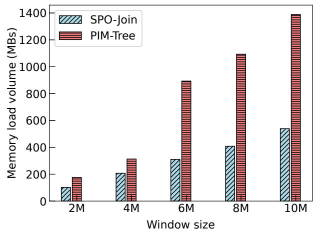


Figure 13: Memory cost

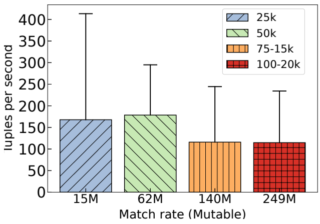


Figure 14: Match rate for mutable window W_M

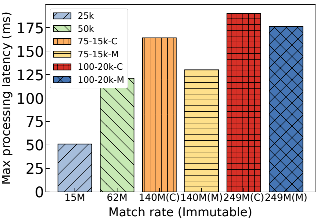


Figure 15: Match rate for immutable window W_{IM}

Chained Index structure. Figure 11a and Figure 11c show the comparison of the event time latency of proposed PO-Join solution and chain index-based (CI) for Q1 and Q3. Figure 11a shows that for the 95th percentile of tuple processing, the proposed PO-Join-based design has 7.0x, 4.2x, 3.7x, 5.5x, 11x, and 11.7x superior performance with increasing slide interval and window size than chain index. Moreover, for 75th percentile; the PO-Join structure has 14.3x, 3.7x, 3.0x, 6.5x, 13.6x, 19x better than alternative. Similarly, the proposed PO-Join structure has a higher performance than the chain index for the 50th percentile, the 23x, 9.19x, 3.2x, 3.8x, 7.5x, and 8.7x. Figure 11c depicts the results for a two-way cross join with increasing sliding interval and window sizes. Results show that for the 95th percentile of tuple processing, the proposed SPO-Join has 14.1x, 11.9x, 21.5x, 31x (sub-slide interval), and 51x (sub-slide interval) superior performance than chain index solution. Similarly, for the 75th percentile, the performance improvement of the SPO-Join structure is 12.2x, 11.4x, 23.9x, 61x (sub-slide interval), and 74x (sub-slide interval) better than the traditional chain index. Similarly, the 50th percentile of

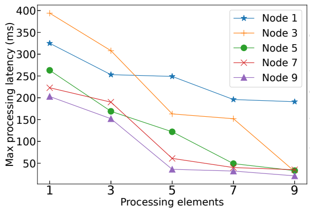


Figure 16: Increasing nodes

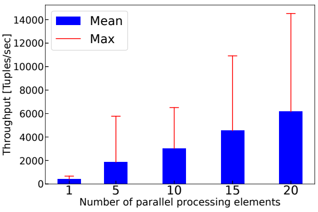


Figure 17: Increasing PEs

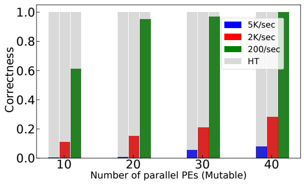


Figure 18: Correctness for mutable window W_M

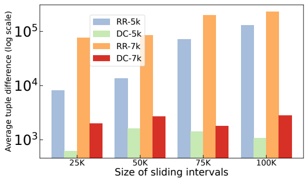


Figure 19: Extra tuples in immutable window W_{IM}

data also depicts the better performance of the proposed SPO-Join structure than the alternative.

In the chain index, new tuples search multiple sub-indexes that increase the search time and the tuples queue waiting time on *PEs*. However, for the PO-Join structure, the sorted data in contiguous memory and efficient computation of logical operation reduce the result set scanning time.

Partitioning based stream join. Figure 11b depicts the throughput results for stream SPO-Join with split join (SJ) and broadcast hash join (BCHJ) algorithms against Q3. These algorithms are also known as nested loop join (NLJ). The split join partitions to the downstream processing instances in a round-robin way, whereas BCHJ broadcasts a single tuple to all PEs. Moreover, PEs employ nested loops to evaluate the queries. It depicts that the proposed join structure has 71x and 32x superior mean throughput for 50K of slide interval and 500K of window size. Similarly, for a 70K slide interval; the proposed SPO-Join has 77.6x and 46x better tuple processing throughput. Additionally, with an increase in slide interval and window size, the performance of BCHJ and SJ is identical. However, the proposed data structure has a 90x time better performance than the alternative. Figure 11d depicts the results of throughput for Q1 with larger slide interval and window size. Results depict that the SPO-Join has 43x superior performance than the alternative. Moreover, the slide intervals do not impact too much on these partitioning schemes.

The partitioning-based schemes employ a nested loop strategy where every tuple is required to expedite the whole list for complete predicate evaluation. Moreover, another operator is required to perform intersection operations after predicate evaluations. However, the SPO-Join uses a B^+ tree for insertion and uses an efficient immutable structure that does not require another level of operator.

Insertion cost. Figure 12 illustrates the difference in insertion cost between SPO-Join, PIM-tree, and B^+Tree. For a window length of 10M with 1M inserting tuples, PIM-tree outperforms SPO-Join with a 1.3x better insertion cost. However, inserting in the mutable part of SPO-Join is 1.15x superior to simple B^+Tree. As the window length and inserting tuples increase, the performance of PIM becomes worse than SPO-Join. The graph shows that SPO-Join has 1.5x and 1.7x better insertion performance than PIM-tree and B^+tree for 50M window length and 5M new tuples insertion, respectively. When a new tuple is inserted into the mutable part of the window, it reduces the cost of insertion. Moreover, SPO-Join avoids the overhead of exploring immutable data structures.

Memory cost analysis. Figure 13 depicts the memory analysis for joining algorithms for increasing count-based sliding windows and cross join query. The results depict the data structure employed by the proposed SPO-Join algorithm consumes an average of 1.5x less memory than the PIM-based join strategy for 2M and 4M sliding windows, however, for larger size windows the memory consumption by SPO-Join performs 2.5x better than the alternative. The SPO-Join only contains indexing data structures for mutable windows. However, PIM join contains many indexing data structures for both mutable and immutable parts of sliding windows.

Match Rate. Figure 14 illustrates the mean and maximum processing throughput for the mutable part of the SPO-Join algorithm with different match rates for the query Q3 against the synthesized dataset. It shows that for a 15M match rate and 25K slide interval, the mean throughput is 167 tuples/seconds, with a maximum of 245 tuples/seconds. Similarly, for a 62M match rate and 50K slide interval, the mean throughput is 179 tuples/second. When the match rate is higher, such as 140M and 249M, the average throughput just drops to 115 tuples/second and 114 tuples/second, respectively.

In Figure 15, the maximum processing latency for the immutable component of the proposed data structure is shown with varying match rates. The graph indicates that for a 15M match rate and 25K slide interval, the maximum processing latency is 51ms, whereas for a 62M match rate and 50K slide interval, the maximum processing latency is 121ms. For higher match rates, the size of the linked list increases, resulting in a maximum processing latency of 164ms and 190ms for 140M and 249M match rates, respectively for scale-out processing. Similarly, processing the PO-Join with an increasing number of threads (scale-up) results in a maximum processing latency of 130ms and 176ms for higher match rates.

Scalability. Figure 16, scalability is measured with an increasing number of nodes for Q3. The maximum processing latency on the first *PE* for a single node is 325ms, while on the 5^{th} *PE*, it is 191ms. With three nodes, these latencies are 394ms with one *PE* and 31ms for 5 PEs. However, as the number of processing nodes increased, the maximum processing latency on *PEs* decreased. Finally, for nine nodes, these latencies were 203ms on the first PE and 21ms for 5^{th} *PE*. Figure 17 shows the results of throughput with increasing *PEs*. Mean throughput changes from 419 tuples/sec to 6167 tuples/sec with an increase from 1 *PE* to 20 *PEs*. Moreover, the maximum throughput increased from 668 tuples/sec to 14519 tuples/sec.

The increasing number of machines distributing the input tuple load to other machines impacts the processing latency of each tuple. Similarly, increasing the number of *PEs* reduces the waiting time for tuples in the queue. For a small number of *PEs*, more slide intervals are added into the immutable data structure on a single PE where new tuples are required to search all intervals for processing.

Correctness and false positive. Figure 18 shows the correctly computed join result for the mutable part of SPO-Join for Q3 using the synthesized dataset. The results show that for higher

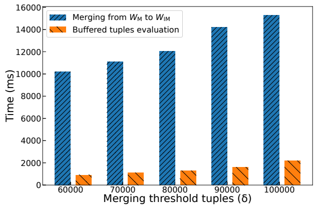


Figure 20: Impact of merging threshold (δ)

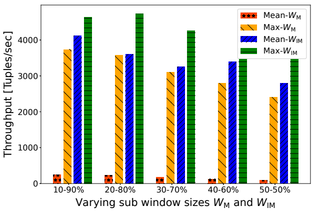


Figure 21: Varying sizes of W_M and W_{IM}

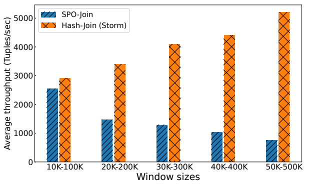


Figure 22: Throughput for equi join

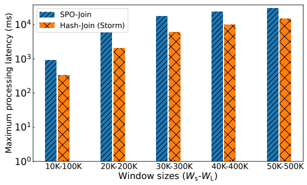


Figure 23: Maximum processing latency for equi join

insertion rates, such as 5,000 tuples/sec and 10 downstream *PEs* of the logical operator, only 0.3% of tuples are correctly computed in their join results. However, increasing the number of *PEs* increased the accuracy in identifying correct tuples. This also does not guarantee 100% correctness for the high insertion rate of tuples as shown for 40 *PEs* and 500 tuples/sec, where the correctness approaches only 8%. However, hash partitioning of upstream tuples and the use of a small hash table guarantee 100% correctness.

Figure 19 shows the tuple differences between different *PEs* of PO-Join operator for both of our proposed window state management strategies for high input data rates. For a 25K slide interval and 5,000 tuples/seconds, the average difference between the tuples among first *PE* to others is 13x better for distributed cache-based (DC) tuples state synchronization than for the partitioner-based round-robin (RR) scheme. This means that a new tuple can produce join results on the expired tuple of the sliding window (false positive). Similarly, for 7,000 tuples/sec, this difference is 38x improved than the alternative. Moreover, for 100k slide intervals, this difference is 82x and 94x. The use of a distributed cache among processing instances can reduce the false positive rate of tuple processing for a sliding window.

Figure 20 illustrates the impact of merging with varying merging threshold (δ) for Q3. The findings indicate that the SPO-Join structure takes between 10 and 15 seconds to merge tuples within slide intervals ranging from 60K to 100K, merging threshold δ. This time includes the computation of the permutation array, network cost, and construction of the PO-Join structure on its respective processing element (PE). Additionally, Figure 20 also demonstrates the evaluation of buffered tuples on specific *PE* of the PO-Join operator. The results indicate that these tuples are evaluated efficiently and only take between 1 to 2 seconds for δ from 60K to 100K. The PO-Join operator is specifically designed to efficiently compute the inequality operator and optimize the logical operations.

Figure 21 shows tuple processing throughput with varying sub-windows (mutable W_M and immutable W_{IM}) for a fixed 1M tuples window length W_L against Q3. For 10-90%, the 100K tuples belong to the W_M and 900K to W_{IM}. Results depict that the maximum tuple processing throughput for W_M is 4,124 tuples/sec, however, its average processing throughput is 249 tuples/sec, similarly, it is 2,800 tuples/sec and 96 tuples/sec for 50-50% sub-windows. New tuples are always inserted into W_M. Initially, the size of the mutable window W_M is smaller so the tuple processing throughput is higher. Incoming tuples require less waiting time for evaluation; however, with an increase in the size of tuples, processing throughput decreases. The immutable window W_{IM} holds only the fixed batches of past mutable windows on distributed *PEs*. The insertion cost for tuples is negligible (only during merge).

Figure 22 shows the overall throughput of SPO-Join and hash join (Apache-Storm) using 10 *PEs* for Q1 with equality predicate operators instead of inequality on a uniformly distributed synthetic dataset. The results indicate that the average throughput of hash join is only 1.14x superior to SPO-Join for a 10K slide interval. However, for larger slide intervals, the overall throughput for SPO-Join decreases, as depicted that for slide-interval of 50K, hash join is 6.8x better throughput than SPO-Join. Similarly, Figure 23 shows their maximum processing latency for equi-join. It is depicted that hash-based join is 2.7x and 2.02x better than SPO-Join for a 10K-100K and 50K-500K sliding window. It is observed that the maximum latency of SPO-Join is slightly improved with increasing sliding windows, which is due to its higher number of tuples processed by the PO-Join operator of SPO-Join. The hash join has a negligible searching and insertion overhead O(1), with the only overhead being the removal of tuples from the slide interval. For larger slide intervals and window lengths, the tuples removal threshold arrives late, allowing for more tuples to be processed.

6 CONCLUSION

We explored how to efficiently index the contents of sliding windows to support inequality join queries in a streaming environment. State-of-the-art index-based solutions can experience significant overhead when updating indexes for larger sliding windows. To address this challenge, we propose a novel methodology that leverages a sorted array-based solution that outperforms traditional indexing strategies. We propose a novel approach that exploits both a mutable and an immutable data structure. New tuples are inserted into the mutable B$^+$-tree (good for fast insertion), which is merged into an immutable PO-Join structure (good for fast search) depending on the merging threshold δ. The PO-Join structure maintains most tuples in the sliding window, while only new tuples are added to the mutable part. We also propose a novel algorithm to build the PO-Join structure in the streaming scenario efficiently. Finally, we conducted extensive experiments to evaluate the performance of our proposed solution. Our approach outperforms other index-based solutions, making it a promising solution for stream inequality join queries.

ACKNOWLEDGEMENT

We gratefully acknowledge the support provided by Rif.PA 2023-20467/RER and FAR_DIP_2023_DIEF-SIMONINI, under project numbers CUP E83C23002540002 and CUP E93C23000280005.

We acknowledge the late Prof. Jorge-Arnulfo Quiané-Ruiz for his invaluable contributions and support to this research. His passing last year was a great loss, and his legacy continues to inspire this work.

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceeding of the VLDB Endowment* 5, 10 (2012), 1064–1075.

[2] Apache Flink Documentation: Joins in SQL Queries. 2024. . https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/sql/queries/joins/

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the International Conference on Data Engineering*. IEEE, 362–373.

[4] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To partition, or not to partition, that is the join question in a real system. In *Proceedings of the International Conference on Management of Data*. 168–180.

[5] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528.

[6] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the International Conference on Management of Data*. 2471–2486.

[7] Buğra Gedik, Rajesh R Bordawekar, and Philip S Yu. 2009. CellJoin: A parallel stream join operator for the cell processor. *The VLDB journal* 18 (2009), 501–519.

[8] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data* 7, 2 (2016), 299–312.

[9] Bingsheng He and Qiong Luo. 2006. Cache-oblivious nested-loop joins. In *Proceedings of the International Conference on Information and Knowledge Management*. 718–727.

[10] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 grand challenge. In *Proceedings of the International Conference on Distributed Event-Based Systems*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 266–268.

[11] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *Proceeding of the International Conference on Data Engineering*. IEEE, 1507–1518.

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

[13] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and scalable inequality joins. *The VLDB Journal* 26, 1 (2017), 125–150.

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

[15] Thomas Kriechbaumer and Hans-Arno Jacobsen. 2018. BLOND, a building-level office environment dataset of typical electrical appliances. *Scientific Data* 5, 1 (2018), 1–14.

[16] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the International Conference on Data Engineering*. IEEE, 38–49.

[17] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-optimal distributed band-joins through recursive partitioning. In *Proceedings of the International Conference on Management of Data*. 2375–2390.

[18] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the International Conference on Management of Data*. 811–825.

[19] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. {SplitJoin}: a scalable, low-latency stream join architecture with adjustable ordering precision. In *USENIX Annual Technical Conference*. 493–505.

[20] Alper Okcan and Mirek Riedewald. 2011. Processing theta-joins using mapreduce. In *Proceedings of International Conference on Management of data*. 949–960.

[21] Jun Rao and Kenneth A Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the International Conference on Management of Data*. 475–486.

[22] Maximilian Reif and Thomas Neumann. 2022. A scalable and generic approach to range joins. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3018–3030.

[23] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.

[24] Ibrahim Sabek and Tim Kraska. 2023. The case for learned in-memory joins. *Proceeding of the VLDB Endowment* 16, 7 (2023), 1749–1762.

[25] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel index-based stream join on a multicore cpu. In *Proceedings of the International Conference on Management of Data*. 2523–2537.

[26] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed stream KNN join. In *Proceedings of the International Conference on Management of Data*. 1597–1609.

[27] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the International Conference on Management of Data*. 625–636.

[28] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the International Conference on Management of Data*. 147–156.

[29] Tri Minh Tran and Byung Suk Lee. 2010. Distributed stream join query processing with semijoins. *Distributed and Parallel Databases* 27, 3 (2010), 211–254.

[30] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *The VLDB Journal* 32, 5 (2023), 985–1011.

[31] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. 2023. Nested loops revisited again. In *Proceedings of the International Conference on Data Engineering*. 3708–3717.

[32] Jianye Yang, Wenjie Zhang, Xiang Wang, Ying Zhang, and Xuemin Lin. 2020. Distributed streaming set similarity join. In *Proceedings of the International Conference on Data Engineering*. IEEE, 565–576.

[33] Eleni Zapridou, Ioannis Mytilinis, and Anastasia Ailamaki. 2022. Dalton: Learned partitioning for distributed data streams. *Proceedings of the VLDB Endowment* 16, 3 (2022), 491–504.

[34] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M Grulich, Steffen Zeuch, Bingsheng He, Richard TB Ma, and Volker Markl. 2021. Parallelizing intra-window join on multicores: An experimental study. In *Proceedings of the International Conference on Management of Data*. 2089–2101.

[35] Shunjie Zhou, Fan Zhang, Hanhua Chen, Hai Jin, and Bing Bing Zhou. 2019. FastJoin: A Skewness-aware distributed stream join system. In *IEEE International Parallel and Distributed Processing Symposium*. 1042–1052.