Efficient Distributed Stream Inequality Join

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

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

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

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 (all existing methods employ a variation of B^+tree).

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. Further, our proposed method is designed to be efficiently executed with distributed stream processing engines—we provide an open-source implementation for Apache Storm. Our experiments on real-world and synthesized datasets suggest that the proposed SPO-Join exhibits superior performances compared to state-of-the-art index-based stream inequality join solutions.

PVLDB Reference Format:
Adeel Aslam, Giovanni Simonini, and Kaustubh Beedkar. Efficient Distributed Stream Inequality Join. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

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 continuously maintain and update data structure employed to facilitate finding the matching tuples [23]. To this end, several distributed stream join models and algorithms have been proposed [5, 11, 16, 32].

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. For instance, consider the following example in the context of real-time energy consumption monitoring.

Example (Power monitoring in data center). *A data center comprises many IT devices including servers, storage devices, and network components that connect these devices to each other. All these devices consume electrical energy and dissipate heat that requires proper cooling infrastructure to avoid system failure and ensure the up-time of jobs. According to the report by the International Energy Agency (IEA), the power consumption by data centers is 240-340Twh nearly 1-1.3% of global energy consumption. It is also gradually increasing as a significant environmental problem* [4].

Ellen, a support and network engineer for a cloud service company that has two data centers (DC1 and DC2) at different locations. DC2 is slightly bigger in size and has more racks than DC1, however, it has the same cooling infrastructure such as Computer Room Air Conditioners (CRACs). Ellen makes the rescheduling decision of jobs between data centers depending on their racks (RP) and cooling infrastructure (CP) power profiles. He initiates a real-time query Q 1 that filters only those real readings where racks power consumption by DC1 is less and more cooling power usage than DC2 within a specified interval (C shows the slide interval and D represents the size of the sliding window.).

Q 1: Real-time power analysis of racks and cooling infrastructure for data center

```
SELECT  DC1.RP, DC1.CP, DC2.RP,
DC2.CP FROM PowerConsumptionData WHERE
DC1.RP < DC2.RP AND DC1.CP > DC2.CP
WINDOW AS (SLIDE INTERVAL 'C' ON COUNT 'D')
```

In this scenario analysis Q 1, we compare DC1 with DC2 in terms of rack power consumption and cooling infrastructure power usage. The query involves two inequality predicates and one intersection operation. Ellen's requirement is that the processing system should quickly compute the results for every real-time reading from either DC1 or DC2. To achieve this, he exploits existing algorithms and techniques for this window-based stream inequality join query in a distributed environment [27].

However, this type of join have received very little attention in the literature [23], and the state-of-the-art inequality join algorithm

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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

for batch processing has severe limitations if adapted to work with streaming data [12]

Existing inequality join algorithms include indexing with B$^+$tree and CSS tree. The CSS search operation is slightly better than B$^+$tree because it eliminates several pointers and uses relative indexing. However, the insertion cost increases due to block packing and more index reconstruction for every new insertion. The IE-Join [12] works on the principle of permutation and offset array. The search procedure is more optimal for complex queries that include inequality predicates and intersection operators. However, the permutation and offset array calculations exhibit a quadratic time complexity, which poses a significant bottleneck for the streaming environment.

There are some research studies that focus on optimizing the stream inequality join process using fundamental algorithms. Najefi et al. [17] utilize a split-join technique for multi-core systems, where they insert data into the cores of hardware in a round-robin fashion. They broadcast a new streaming tuple to all cores to facilitate probing, leveraging a nested loop for join evaluation. Lin et al. [16] suggest a chain index-based solution for distributed stream join. It consists of a linked set of balanced search tree (B+tree) sub-indexes named active and archive sub-indexes. The insertion of a new tuple is only possible into an active sub-index, and tuple evaluation is performed on all sub-indexes. Shahvarani et al. [22] propose a new technique for stream inequality join queries called PIM-tree. This design comprises a range-based set of insert-efficient mutable data structures (B$^+$tree) and a search-efficient immutable data structure (CSS-tree). The insertion of a new tuple is only possible into a mutable structure while it is probed into both designs.

We also conduct an empirical study on query Q1 using a synthesized dataset with a match rate of 250 million tuples. Our observation showed that the IE-Join data structure had a biased search performance, taking 5.3x, 4.65x, and 21.25x less computation time than the B$^+$tree, CSS tree, and nested-loop join algorithm, respectively. Also, the insertion cost of B$^+$tree was 2x better than the CSS-tree and IE-Join algorithm. However, in the streaming case, the insertion operation has a similar frequency to the search operation. Therefore, the IE-Join technique for stream inequality join is not optimal for streaming.

After this analysis, Ellen concluded that using split join is quite costly for smaller-size worker nodes or cores. The nested-loop join used by split join had an insertion cost of O(1) *for a new tuple. However, the search cost increased with the increasing size of the sliding window. Ellen was not happy with the chain index-based solution either. Adding a new tuple required scanning through linked sub-indexes and leaf nodes of these index data structures, which required many memory scans [23]. Additionally, Ellen did not find an efficient state management strategy for the contents of the sliding window among processing elements of the stream processing engine. Ellen liked the design structure of the PIM tree, but it was too complex for Q 1. This design does not provide a solution to handle intersection operations, and a naive solution requires two intersection operators, one each for mutable and immutable in-memory data structures. Moreover, it also needs an efficient data provenance strategy for distributed stream processing. However, after observing the probing performance of IE-Join, Ellen looked for a solution where most of the tuples of the streaming window were evaluated by the IE-Join design instead of the CSS tree. This approach eliminated intersection overhead and did not require an extra provenance strategy for processing guarantees. However, this approach comes with certain challenges.*

Challenges: The most attractive design is the use of IE-Join data structure [12]. 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 with correct computation of permutation and offset array. Ellen employed a two-way structure inspired by the PIM tree design to get the benefits of IE-Join design. The sliding window is divided into two parts: mutable B$^+$tree data structure and immutable structure that is named as permutation and offset-based join (PO-Join). A new tuple was evaluated against both in-memory data structures for completeness. However, considering Q 1, which consisted of three operators - two predicates and an intersection - the mutable structure required efficient processing of predicate operators and intersection. Additionally, in a distributed setting and high-speed insertion rate, it also requires an effective data partitioning and data provenance strategy for correctness. Merging a mutable tree into an immutable structure posed difficulty since the IE-join data structure required the computation of permutation and offset arrays that are quadratic in time complexity. Moreover, in distributed stream processing (DSP), where sliding window contents are held by different processing elements, maintaining a consistent state of the sliding window among all processing elements of the DSP system is a challenging task.

Solution: In this study, we present a two-way data structure consisting of a B$^+$ tree (indexing data structure) and a linked set of PO-Join structures called *stream permutation and offset-based distributed stream inequality join*. The B$^+$ tree structure accepts new tuples only for insertion. When the number of tuples reaches a certain threshold, the indexing data structure is merged and a PO-Join design is created for the downstream worker processes, which are assigned in a round-robin manner. During insertion, a tuple is evaluated in a mutable tree while it is simultaneously broadcasted to all processing elements holding the PO-Join structures of the sliding window. Similar to the chained index, a coarse-grained tuple removal strategy is adopted, where an entire PO-Join structure is removed from memory instead of a single tuple removal. In summary, the proposed approach provides a novel solution for efficient query processing in sliding window scenarios.

This study proceeds in such a way; Section 2 provides preliminary knowledge about stream inequality join, moreover it also sheds light on basic approaches used for SPO-Join design. Section 3 provides the overview of the proposed SPO-Join strategy. However, in Section 4 we provide a detailed discussion on various aspects of SPO-Join. In Section 5 and Section 6, we provide a detailed description of the experimental setup and result. Section 7 provides related work, and finally, Section 8 concludes this study.

2 PRELIMINARY

This section provides definitions and descriptions used in stream inequality join for distributed stream processing.

2.1 Stream Inequality Join

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

Inequality join: Let's assume two data stream $R = \{r_1, r_2, r_3, \ldots\}$ 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; $R \bowtie_\theta S$ or from any single stream $R_i \bowtie_\theta R$. So, tuples from both streams involve two-way join, where a tuple r_i from stream R needs to be evaluated against S tuples, similarly, an analogous procedure with the opposite predicate is required for a new tuple s_j from stream S. $R \bowtie_\theta S = \{(r_i, s_j) \mid r_i \in R, s_j \in S, r_i \bowtie_> |S| \vee s_j \bowtie_< |R|\}$. This type of inequality join queries is also known as high selective queries.

Sliding window: The bounded set of streaming tuples is termed as sliding window $W = \{t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+L}\}$; where L is the total items in the sliding window. It can be classified into two primary categories such as count-based W_c or time-based W_t tuple boundaries. In the sliding window; on the arrival of a new tuple t_i, obsolete tuples t_n where $n = \{1 \leq i \leq n\}$ are discarded from the predefined size of the window. Moreover, t_n can also be defined as the slide interval W_S whereas W_L represents the total length of the sliding window. (In this study, our focus is solely on the count-based sliding window. However, it is noteworthy that the same semantic can be applied to the time-based window)

2.2 Inequality Join Solutions

In this subsection, we provide a description of algorithms and techniques used in our stream inequality join.

Indexing and search with B$^+$ tree: B$^+$ tree is a self-balancing data structure used for indexing the data items for efficient retrieval of data. In B$^+$tree, the data is stored in the leaf nodes of the tree, whereas the internal nodes act as indexes to make the search more efficient. The internal nodes hold the keys and the pointer toward child nodes. These pointer helps to efficiently determine the child node during the search [16, 23].

For *insertion*, the tuple retrieval process starts from the root node to the leaf node. All items in the leaf nodes are sorted, and after retrieving all relevant nodes the data item is inserted or appended into the leaf nodes of B$^+$tree. Analogous to the insertion; the *search* procedure also follows the same procedure. Moreover, keys are sorted in the leaf node, so performing a range search after finding the relevant key requires *O(log n + k)*. Here, the key is the number of elements in the range; in the worst case *k=n*.

IE-Join: The IE-join procedure comprises two phases: initialization and probing. Initially, the IE-join data is designed for the collected batch data. The first step involves sorting the collected data, followed by constructing the permutation and offset array. A permutation 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 $s_j \in S$. Let us consider Q 1 as a use case with fixed batch data. In this case, Q 1 requires two permutations and two offset arrays.

In the Q 1 use case, a lightweight bit array is used in the probing phase. Initially, all the bit positions are set to 0. The search procedure starts with the first tuple of field $DC1.CP \in DC1$. It first finds its relative position in $DC2.CP \in DC2$ from the offset 2 (O2). Then, it sets all the bit positions of permutation 2 (P2) from the start to the relative position of O2. Afterward, it uses the offset 1 (O1) that contains the relative position of field $DC1.RP \in DC1$ in $DC2.RP \in DC2$ to populate all the join results [12] (Section 3.1). This IE-Join structure is well-suited for fixed data, it is not practical for continuous data. It employs a packed memory array, which is not feasible for larger window sizes and persistent data.

2.3 Distributed Stream Processing System (DSPS)

A DSPS uses a cluster of nodes to process input data in a parallel pipeline fashion, providing high throughput and low latency for tuple processing. The DSPS uses a master-slave architecture, where a new application in the form of a *directed acyclic graph* (DAG) is submitted to the master node. This node then spawns the application towards slave or worker nodes, depending on the job scheduling algorithm used by the master node. In the context of the stream, inequality join processing, three essential components play a pivotal role: 1) processing elements (PEs), 2) data partitioning, and 3) stream join by DSPS.

Processing elements: DAG application comprises many vertices v. Each vertex can have multiple processing elements denoted by $PE_i = \{PE | 1 \geq PE \leq m\}$. These processing elements carry out the actual operation tasks such as joining, filtering, aggregation, etc. Moreover, they are also isolated from each other.

Stream partitioning: When processing data between the vertices v_i and v_j, a partitioning strategy is employed. Hash partitioning is a common method, whereby the selection of the downstream processing element is made by applying the same hash function to the data unit. Other partitioning schemes may also be used, such as 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.

Stream join by DSPS: In the stream joins, data is bounded within sliding windows. Usually, specific data structures are used to hold the items of the sliding window by employing DSPS processing elements. When a new tuple $\{r, s\}$ is introduced, it first probes these data structures through the processing element for query predicate evaluation. Additionally, for complex queries that involve more than one predicate, such as Q 1, multiple vertices of DAG are used for evaluation depending on the number of predicates and operations. The results from each individual processing element require further transformation for completeness of the result that employs data partitioning and a data lineage scheme. Each processing task is independent, and its data structure holds only a subset of sliding window items. Therefore, state management schemes among processing elements are employed for sliding window contents.

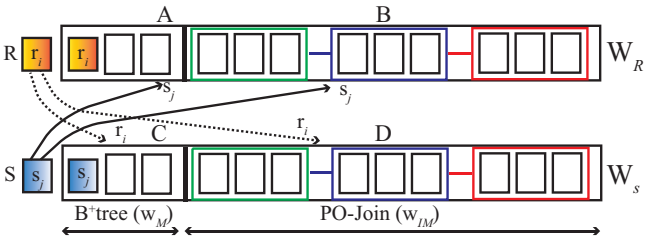


Figure 1: Proposed window-based stream IE-Join

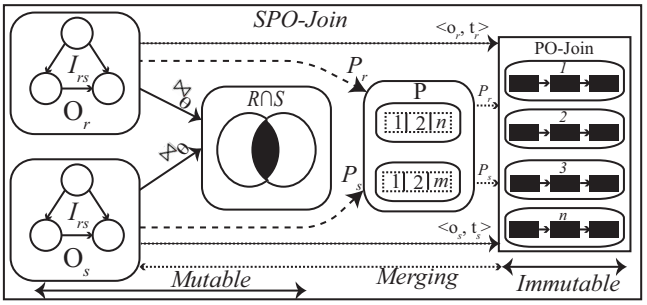


Figure 2: DSPS model for stream inequality join

3 SOLUTION OVERVIEW

In our stream inequality solution, we make use of B$^+$ trees and the IEJoin data structure to index and efficiently probe the tuples in the sliding window of streaming data. In this section, we provide an overview of both the logical and system perspectives of stream inequality joins for DSPS.

The high-level overview of the proposed stream in-equality join solution can be seen in Figure 1. Consider two streams, R and S, each building their own sliding windows W_R and W_S, respectively. Each window is divided into two components: 1) a mutable component (W_M) with streaming contents A for W_R and C for W_S, and 2) an immutable component (W_{IM}) with streaming contents B for W_R and D for W_S. The stream inequality 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 stream inequality join, a new tuple r_i or s_j from either stream must evaluate all components of the opponent stream for complete predicate evaluation.

Let's consider a tuple, denoted as 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 $(r_i \bowtie_\theta C) \cup (r_i \bowtie_\theta D)$. An analogous procedure is applied for tuple s_j from stream S, whose join result is $(s_j \bowtie_\theta A) \cup (s_j \bowtie_\theta B)$. This tuple is inserted into the mutable component of W_S. 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 in the sub-component of the sliding window. Normally, for a smaller slide interval of window W_S, we use it as the same as that of the slide interval or a combination of many slide intervals as a merging threshold (δ). However, for a larger sliding interval, we subdivide the slide interval into sub-intervals to avoid large merging overhead, as detailed in Section 4. 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} of the streaming window. This operation frees up space for new tuples in W_M. As analogous to the chain index [16] and PIM join [23], a coarse-grained tuple removal strategy is applied to the W_{IM} component of the sliding window, where the old index from the linked set of PO-join structure is removed for tuple removal threshold. In Figure 1, the red box represents an old index, which is removed from the window after the new merge but depends on the tuple-removing threshold.

To process stream inequality joins in a distributed and parallel manner, we use the system model illustrated in Figure 2. For complex queries like Q 1, which involve two opposite streams R and S and their intersection, we divide the execution into three components: mutable (B$^+$tree and Offset), permutation computation, and immutable (PO-join). For the mutable design, we have three operators that perform operations in a pipeline-parallelism manner. Two parallel operators evaluate individual predicates and transform their partial results to the intersection operator using hash partitioning. This operator contains multiple parallel processing elements that can process the partial results of several tuples simultaneously. 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. Proper PO-join structure construction and removal processes are discussed in detail in section 4.

4 PERMUTATION AND OFFSET-BASED DISTRIBUTED STREAM INEQUALITY JOIN

This section covers the implementation of the proposed solution for stream inequality join (SPO-Join). The solution can handle either the same stream R or different streams R and S. Additionally, we will discuss how we can utilize multi-cores to speed up the join procedure in the immutable component of the proposed stream join solution. We will provide a detailed description of the data structure and shed light on the merge process along with the assurances for processing streaming tuples.

The proposed SPO-Join process is represented by Algorithm 1. This algorithm takes in the tuple t_i, the slide interval for the window W_S, and the total length of the window W_L. The total number of processing elements for various operators (operators are shown in Figure 2). The output of the algorithm is the join result. When the tuple is inputted into the system, it is partitioned towards mutable and immutable stream join operators as shown in lines 1 and 2. The stream join process on the mutable component is depicted in line 3 and produces the partial result for the streaming window against the tuple t_i, as detailed in Section 4.1. A merge-interval counter for the count-based window is also incremented on both operators after tuple evaluation, as shown in line 4. In parallel to the stream join in the mutable component, the inequality join process for tuple t_i is initiated on the operators (PEs of the immutable part) of Immutable components (PO-Join operators) as depicted by line 6, and is detailed further in Section 4.3. From line 7 to line 11, Algorithm 1 describes the merging process for the mutable component to the immutable part. This whole process includes offset computation for two-way join, permutation computation, and data partitioning during the merge process as detailed in Section 4.2. Line 12 depicts the creation

of the PO-Join structure to the downstream processing element, and line 13 shows the updating of the sliding window state for slide interval tuple removal. After merging, the merge interval of the mutable component of the stream is re-initiated for other incoming tuples.

4.1 Mutable Component

To process the data efficiently in the proposed stream inequality join, the mutable part uses B$^+$tree indexing data structures, I_R and I_S. The new tuple (t_i) is first indexed into its respective index structure with key-value pair. Once indexed, data relevant to the inequality condition specified is probed from the underlying indexing structure.

Tuple insertion: To understand the insertion process, we utilize our first use case and Q 1. In this case, a tuple t_i comes from data center DC_1 that contains the real-time readings for power profiles of racks and cooling infrastructure. According to Q 1, our DAG consists of two processing elements (PEs) for each operator (>, <). Each operator builds two indexing data structures: I_i for $DC_1.RP$ and I_j for $DC_2.RP$, for streaming tuples (resp., $DC_1.CP$ and $DC_2.CP$)). The next step is to split and partition a new tuple $t_i = \{id, [RP, CP]\}$ into both PEs, where $t_i = \{id, RP\}$ to PE_1 and $t_i = \{id, CP\}$ to PE_2. The t_i tuple is then indexed into I_i in PE_1 and evaluation begins from I_j (resp., PE_2) .

Lookup: Similar to the process of insertion, searching is also done on the indexing data structures (I_R and I_S) at different processing elements. Additionally, an intersection operator is required, where the result from each individual operator is sent for intersection or other operations as shown in Figure 2.

To better understand the lookup process, let's continue with the use-case of Q 1 , and Figure 3 (Here DC1 reading is considered as stream R and DC2 is stream S). For tuple $r_i :<$ id:202, DC1.RP:3000 >, the identified leaf node is N=3300; however, the result set should include all tuples in the leaf nodes where $r_i < DC2.RP$ as depicted by Figure 3.Instead of a hash table, a bit set is introduced, with bits set to 1 where the predicate is true. After predicate evaluation, this bit set is partitioned towards the intersection operator using hash partitioning.

Similarly, for $t_i :<$ id:202, DC1.CP:500 >, where the predicate evaluates $t_i > DC2.CP$, the result set includes all tuples less than N=500. The calculated bit set is forwarded towards the intersection operator using hash partitioning (H(t_i)). Once both bit sets arrive at the intersection operator, an AND operation is performed between the bit sets to obtain the complete 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 to the start or end. A single bit flip in the bit array is $O(1)$, but in our case, there can be m bits, which can result in a cost of $O(m)$ in the worst case. Furthermore, although $O(n)$ is considered the intersection cost of a bit set, modern architectures operate on fixed-size integers (e.g., 32-bit or 64-bit integers) instead of the number of bits set in the operands, making it a constant cost [18].

Algorithm 1: Stream inequality join (SPO-Join)

Input: t_i=$\langle k, v\rangle$; W_L; W_S; PEs$_M$; PEs$_\cap$; PEs$_P$; PEs$_{\text{PO-Join}}$
Output: $R \bowtie_{>,<,\geq,\leq} S$

1 $t_i \longrightarrow$ PEs$_{\text{M}}$
2 $t_i \longrightarrow$ PEs$_{\text{PO-Join}}$
3 Result-M= {(PEs)$_{\text{M}}p \mid t_i \bowtie W_M$} ; // Mutable
4 merged-interval+1; ; // For count based window
5 $\langle t_{id}, \text{Result}\rangle \longrightarrow$ PEs$_\cap$
6 Result-IM={$\forall$(PEs)$_{\text{PO-join}} \mid t_i \bowtie W_{IM}$} ; // Immutable
7 **if** *merged-interval* > W_s **then**
8 Compute offset_array O in PEs$_{\text{M}}$ and partitioned to PEs$_{\text{PO-Join}}$: $ID, \langle \text{O}[pos], t \in W_M\rangle$
9 Partitioned tuples from PEs$_{\text{M}}p$ and partitioned to PEs$_{\text{P}}$: $\langle ID, Tuple\rangle$
10 Compute permutation array P at PEs$_{\text{P}}$;
11 Partitioned permutation array P to PEs$_{\text{PO-Join}}$ $ID, \langle P[pos], t \in W_M\rangle$
12 Insert $\langle ID, (O[pos], P[pos])\rangle$ to Linked-list of $PE_i \in$ PEs$_{\text{PO-Join}}$
13 Update state for W_{IM}: for $W_c \longleftarrow$ merged-interval
14 merged-interval=0;

4.2 Merging to Immutable Component

In the proposed stream inequality join, we utilize the merging of mutable components to an immutable (PO-Join) structure. However, the PO-Join structure consists of sorted data items and their permutation and offset arrays. The key advantage of mutable components is that the leaf node of the indexing data structure provides us with sorted elements. We use these sorted items to compute the permutation and offset arrays. The offset array is necessary for two-way joins, and its computation is performed within the *mutable part* of the processing element. This is because these processing elements encompass both the indexing data structure requirements, for which the offset array is essential. After computing the offset array, the result is partitioned to the PO-Join operator with <id, OO_i>. Moreover, we perform permutation between the same streaming data on different fields that exist on different mutable PEs. Therefore, we introduce an intermediate permutation operator with processing elements (PEsp). These processing elements compute the permutation array and forward it towards the PO-Join structure as <id,P_i>. Permutation and offset array computations are independent of each other so they can be executed in parallel on mutable PEs. For permutation, the tuple is simply partitioned, however, for offset, its location is computed and then passed downstream toward the PO-Join operator.

Permutation computation: The process of computing permutations is outlined in Algorithm 2. The algorithm takes a sorted array of tuples from both fields a and b of the same streaming data R as input and produces an output that provides us with the permutation location of field b in field a. To accomplish this, the algorithm initializes a temporary counter on Line 1 and introduces 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 b from left to right and fills the temporary array with an incremental counter. During this process, the identifier of the field b is considered as an index location in the temporary array. The algorithm then iterates through the opposite field items, a, on Lines 6 and 7.

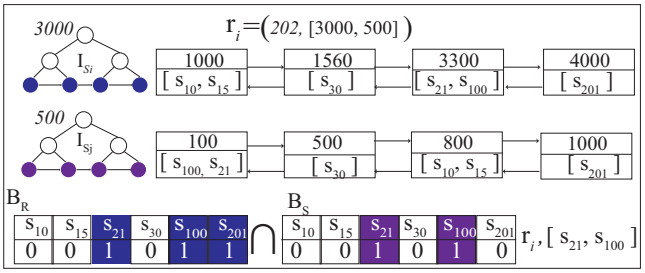


Figure 3: Mutable-part stream join (Example Q 1)

The permutation array is filled with the temporary array with the field *a* identifier and partitioned toward the PO-Join structure.

Offset computation: Algorithm 3 outlines the process for computing offsets, where the input is an indexing data structure for the opposite stream (either stream I_R or stream I_S), and the output is the relative location of data items in I_R with respect to I_S. To begin with, the index position is initialized to 0 in line 1. Starting from line 2, all keys of I_R are iterated. If the index position is 0, the algorithm begins searching for the key k_r in the opposite index data structure I_S, updates the index position with the newly identified position of k_r in I_S, and sets the offset location of k_r to the updated index position O_{kr}. The outer loop then continues to the next key in I_R, as explained in line 3 to 5. Similarly, if the 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 line 8 to line 14. If the key $k_s \in I_S$ is greater than or equal to the key $k_r \in I_R$, then a new offset position is updated, and this position is set as the offset position for k_r. The algorithm then moves on to the next key in I_R, as shown by line 9 to 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 in I_S, as depicted by line 14 of Algorithm 3.

4.3 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, that requires evaluation, the total linked PO-Join data structure, and the number of cores available in the computing node. We ensure during partitioning that each computing node holds only one processing element for the immutable part. However, this processing element can employ many linked sets of PO-Join data structures.

In line 1, we initialize the number of threads equal to the number of cores available in the node. However, another choice could be to initialize these threads with the size of the linked list. This process could increase the overhead of more context switching

Algorithm 2: Permutation computation

Input: Sorted $R_a[Tuple]$, Sorted $R_b[Tuple]$
Output: Permutation-Array (P)[]

```
1  counter ← 1
2  tmp[] ← 0 ;                    // Initialize an empty array
3  for ID_i ∈ R_a do
4  |   tmp[ID_i]= counter
5  |   counter++
6  for ID_j ∈ R_b do
7  |   P[j] = tmp[ID_j] ; // ID remains same among fields
   |    of tuple
8  tmp[] ← null;
```

Algorithm 3: Offset computation

Input: *Index-tree*(I_R), *Index-tree*(I_S)
Output: Offset-array $[r_i, s_i]$ (O_i)

```
1  Offset-index ← 0
2  for k_r ∈ I_R do
3  |   if Offset-index==0 then
4  |   |   Offset-index ← I_S.search(k_r)
5  |   |   O_kr ← Offset-index
6  |   |   continue;
7  |   else
8  |   |   for k_s[Offset-index] ∈ I_S to k_s.Length do
9  |   |   |   if k_s ∈ I_S ≥ k_r then
10 |   |   |   |   Offset-index ← k_s[pos] ; // offset index
   |   |   |   |    is updated with new position
11 |   |   |   |   O_kr ← Offset-index
12 |   |   |   |   break;
13 |   |   |   else
14 |   |   |   |   O_kr ← k_s.Length+1
```

Algorithm 4: PO-Join: t_i evaluation on PE_i

Input: t_i=⟨k_i, v⟩; Linkedlist⟨PO-Join$_{1\rightarrow n}$⟩; |cores|; type [self, two-way]
Output: $R_{IM} \bowtie_{>,<,\geq,\leq} S_{IM}$

```
1  Initialize-threads ← |cores|
2  Current-index ← 0
3  while true do
4  |   lock ;     // Ensure multiple threads do not get
   |    the same [PO-Join]
5  |   if Current-index < Linkedlist.size() then
6  |   |   PO-Join ← Linkedlist.get(Current-index);
7  |   |   Current-index++;
8  |   else
9  |   |   break ;          // all elements are accessed
10 |   unlock
11 |   if type is Equal To 'self' then
12 |   |   t_i ⋈ PO-Join_self ;  // explained with Figure 4
13 |   if type is Equal To 'two-way' then
14 |   |   t_i ⋈ PO-Join_two-way ;        // explained with
   |   |    Figure 5
15 Update the W_Cim for PE_i
```

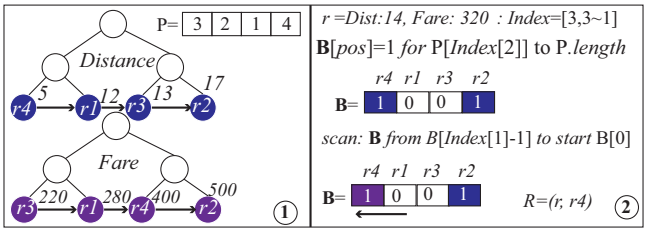


Figure 4: Self join Process with two conditions (Example Q 2)

between threads when the available cores are limited in 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 index of the linked list. In line 5, we check the size of the linked list and ensure that all elements of the linked list are assigned to threads. Each thread performs the join operation on a new tuple t_i of the linked list depending on the type of join, as depicted by lines 11 to 14. Moreover, the self-join and two-way join process is explained in Figure 4 and Figure 5 with examples. Similarly, line 15 indicates the updating of the sliding window counter.

Self-join: Figure 4 showcases an instance of a complex self-join process for a new tuple r_i in Q2. Khayyat et.al [12] has defined the process of permutation array-based self-join, but this is solely for batch data and does not include a way for new tuple evaluation using a permutation array. For self-join, only the permutation array is needed between field a and field b of streaming data. The sorted data from the leaf node of the mutable component is partitioned to the PO-Join operator, and it does not require an intermediate node to perform computation. The entire self-join process is explained with the help of an example of Q 2 with several steps. ① We have sorted data items, a permutation array, and an empty bit set (B) with respect to the field a (Distance). ② A tuple, denoted as t_i=<r_i, [$Dist$: 14, $Fare$: 320]>, is first split into two fields and then a binary search is performed on the sorted arrays of both fields. The resulting index values are Index=[3,3~ 1], where the symbol ~indicates that the value 320 is not actually present in the sorted array of the second field, i.e., field b. ③ From position P[index[2]] to the end of the bit set, set all bits to 1. In this case, we set index 1 and index 4 to be true. ④ Scan the bit array from B[index[1]-1] to B[1]. During scanning, identify all 1s and add them to the result set. The procedure for bit scanning is changed based on the predicate operation as detailed by [12].

Two-way join: Figure 5 depicts the process of a two-way stream inequality join. To determine the relative position of tuple t_i of stream R in stream S, an Offset array is required. In a stream join, the tuple can come from both stream R and stream S. Therefore, we have provided descriptions for both tuples against Q1 along with an example. ① Initialize an empty bit array for stream S upon receiving a new tuple <r_i, $Rack$: 1400; $Cooling$: 3000> from stream R (DC1). ② To find the relative location of a tuple in an offset array, perform a binary search on the offset array O2 for the field b with cooling value 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 the value 2 from the offset array at the position found minus one. ③ Set bit position 1 for permutation array P2 from start to index O[2]=2. In this example, locations P[1]=1 and P[2]=3; now bit positions 1 and 3 should be 1. ④ Perform a binary search for the field a tuple in the Offset array O1. Once you've located the position, start scanning the bit array from O1[pos] to the end. For example, if you perform a binary search for rack 1400 and it returns 1800, you should consider location 3 as the starting point and scan the bit array until the end. All identified 1's in the bit array are the result set. In this case, the result set for DC1 tuple is s3. ⑤ To efficiently evaluate the new tuple from stream S (DC2), we can make use of binary search on the existing offset array 2 and set the permutation array 1 location in the bit set. Figure 5 shows the procedure to evaluate the tuple from the opposite stream. The tuple <s_i, $Rack$: 2100; $Cooling$: 2700> is an example of this process. Updating the bit set and scanning the tuple can be changed based on the predicate value. The evaluation of join for PO-Join with different predicates can be found in [12] Section 3.2.

Tuple evaluation during merge operation: When a batch interval of tuple (δ) arrives in the mutable part of the PO-Join operator, a flag tuple is sent to the selected downstream PE to initialize 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 the PE, the tuples from the queue start probing the newly merged PO-join structure until the queue becomes empty.

Time complexity: In order to create a permutation array, we use temporary space with a size of $O(k)$ to hold the tuple id of any field. Additionally, assigning an index value to the tmp[] requires $O(n)$ time. Similarly, it takes $O(n)$ time to obtain the permutation location of the field b (DC1.CP) in the field a (DC.RP). Here, n represents the total tuple from stream R. The total time complexity for the permutation array is $O(n + n)$, which can be written as $O(2n)$.

We utilize the sorted array of the leaf nodes of index data structures. The first key from I_R requires $O(\log m)$ to find its location in the leaf node of I_S and serve as a reference point for other tuples from I_R. 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 element in I_R so, the total complexity can be written as $O(n + m)$. For the probing phase, we require two binary searches on sorted data items. The time complexity for searching a tuple is $O(\log n)$. Moreover, filling the bit array against the permutation location has a constant cost of $O(1)$. Additionally, scanning the bit array from right to left or left to right takes $O(n)$ in the worst case. So, the total probing cost can be written as $O(\log(n)+n)$. Here, the binary search between opposite fields of data is independent, which allows for parallelization.

4.4 Data Partitioning

Data partitioning is crucial for DSPS performance. Our stream inequality join uses different partitioning schemes for distributed processing levels.

When dealing with mutable components, the results of the predicate are divided using hash partitioning and sent downstream to

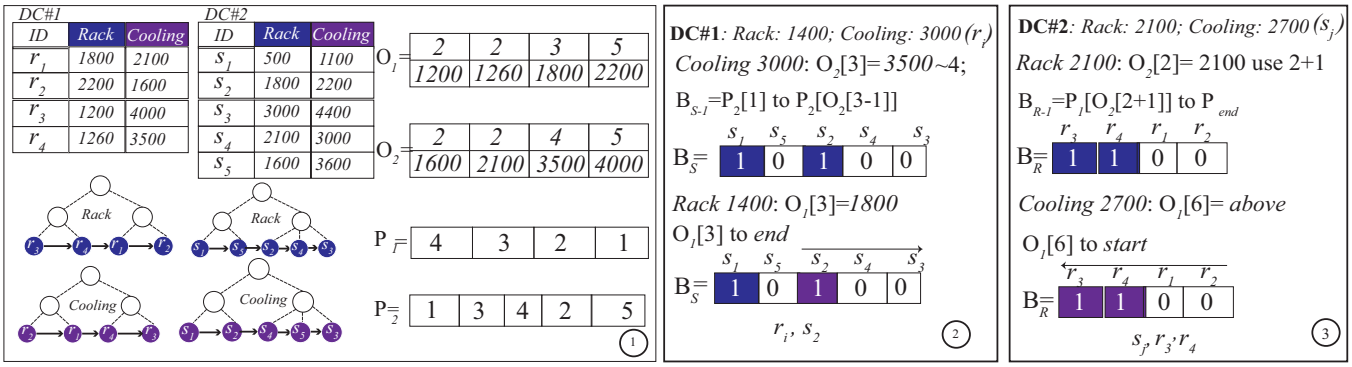


Figure 5: Two-way join (Example: Q 1)

the intersection operator. This ensures that the predicates for each tuple t_i that come from different mutable processing elements (PEs) are sent to the same processing instance for the intersection.

During the merging process, two arrays - permutation array and offset array - are required to send tuples and index positions to the PO-Join operator PEs. A data partitioning strategy is needed to distribute this data. Two scenarios can arise:

1) *Self-Join*: In this case, only the permutation array is required, which is computed on the PO-Join operator. To equally distribute the slide interval load among PO-Join processing elements, a round-robin data partitioning strategy is adopted from the mutable operator to the PO-Join operators.

2) *Two-way Join*: In this scenario, the offset array is distributed to the downstream instances in a round-robin way. However, the permutation array requires intermediate PEs. First, the data is streamed downstream from mutable operators to the intermediate node using direct partitioning to the dedicated PEs for computing the permutation array. From the dedicated PEs, the data is partitioned using a round-robin scheme towards PO-Join processing elements to equally utilize the instances of PO-Join operators. Moreover, the offset array already adopts a round-robin strategy.

4.5 State Management

In the DSPS using sliding window stream join, different processing elements are assigned to process subsets of the sliding window data, which are spawned among various computing nodes. In our implementation of stream inequality join, we employ multiple processing elements for immutable components. To manage the state of the sliding window, particularly for count-based (as time-based is straightforward), we use three different strategies.

Firstly, the selection of downstream processing components depends on the ratio of W_L to W_S, denoted as $PE_{\text{PO-Join}}$. 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 processing element, it signals that the current sliding interval has expired and should be removed from the data structure.

Secondly. When using a larger sliding interval, the approach described above can increase the cost of merging. To solve this problem, we propose dividing the sliding interval W_S with downstream processing elements ($PEs_{\text{PO-Join}}$), which are user-defined.

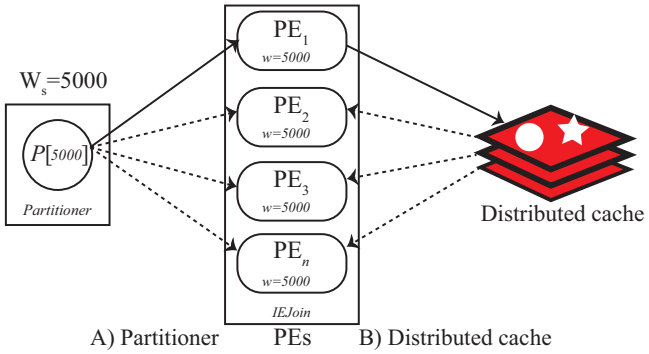


Figure 6: State-management for W_S

In this case, when we send the permutation array downstream to PO-Join PE, we also send the size of merging tuples to all other PEs, except the previous one. This is depicted in Figure 6 (A-left). For every new tuple evaluation in the PO-Join data structure, we check the tuple removal condition, but only remove the first index of the linked PO-Join structure from the PEs. However, this procedure can result in extra tuples in the result set for high input data rates.

Thirdly, to simplify the communication between processing elements (PEs), we have adopted a new strategy that uses a distributed cache. 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 tuple, the sliding window updating procedure is applied to remove the first index of the linked PO-Join data structure from the processing element.

4.6 Processing Guarantees

Mutable: In SPO-Join, it is possible to have multiple indexing data structures within the mutable part. An example of this can be seen in Q 1, where there are four indexing data structures and each new tuple uses two of them for evaluation. However, incorrect results can occur during the final accumulation of the results due to high input data rates or uneven distribution of streaming data among indexing data structures. To ensure that every input stream produces accurate results, we utilize two strategies.

To ensure that tuples with the same ID are processed by the same processing element from the pool of PEs, we use hash partitioning. However, hash partitioning may sometimes result in incorrect outputs if different keys are processed by the same processing element, especially when the intersection PEs are limited. 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 bitset operation can be performed for final result accumulation, and the key is removed from the hash table. Otherwise, it stores the key until another tuple with the same ID arrives.

Immutable: When merging tuples from the mutable part and computing the permutation and offset array, there may be data integrity issues if the input tuple rate is high or if there is a different sliding interval. To solve this problem, each batch of the mutable component (permutation or offset array) is assigned a separate index id (ID). In the PO-Join operator, a smaller hash table is maintained that holds these batches along with their respective ID. For Q 1, 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 EXPERIMENTAL SETUP

In this section, we provide a detailed description of the evaluation metrics, dataset, and experimental setup used to analyze the performance of stream inequality join algorithms.

5.1 Metrics

In this subsection, we will introduce each performance metric used to evaluate the PO-Join algorithm.

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

Latency: For this study, we have chosen to analyze two types of latency: event time processing and processing latency [10]. The event time latency refers to the time taken for a tuple to be completely processed, from the moment it enters the 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 t_i. On the other hand, the processing latency refers to the time taken for a tuple to be processed from the moment it enters the algorithm until it is finally processed. It is important to note that the event time latency also includes the processing latency.

Match rate [23]: The data stream consists of an unbounded sequence of independent tuples. These tuples are further organized into windows for stream joining, with each window having a dynamic matching rate for every new tuple that satisfies the query's predicate conditions. We varied the matching rate for each sliding window syntactically and observed the performance of the proposed join algorithm.

Scalability: Scalability in a system usually refers to its ability to handle an increasing workload. However, 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) [23]. We observe the scalability with an increasing number of processing elements and an increasing number of nodes.

Completeness: The term refers to the accuracy and correctness of the result of join computation for a sliding window on each tuple. We measure the correctness of the mutable part in terms of tuple processing by various processing elements. Similarly, we evaluate the impact of different state management techniques on the completeness of PO-Join under varying insertion rates.

5.2 Dataset Description and Queries

This section outlines the datasets and queries used to evaluate the proposed SPO-Join algorithm.

We used two real-time datasets and one syntactic dataset. The dataset descriptions are also summarized in Table 1.

- **Real datasets:** The New York taxi data set is used in the DEBS Great Challenge of 2015 [9]. It consists of geospatial data related to 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. In addition, we also utilize the building-level office environment dataset known as BLOND. This dataset contains readings of current and voltage for electric appliances in a building. We used a subset of the BLOND-250 dataset that consists of 50 days of readings taken every hour.
- **Syntactic data set**: We use a synthetic data set created at runtime by the DSPS engine for performance analysis with different match rates for tuples.

We use two types of queries: self-join and two-way join. For the New York taxi dataset, we use a self-join query that includes predicates on distance and fare. The resulting tuples should satisfy both predicates, as shown in Q 2. Similarly, for the two-way join, we utilize the BLOND dataset and adopt Q 1 as a use-case. To calculate the power for data center racks and cooling infrastructure, we use various files of current (I) and voltage (V) readings. For the synthetic data set, we use the same queries for both self and two-way joins, but with synthesized values.

Q 2: Real-time analysis for online taxi trip

```
SELECT COUNT(*) FROM NYC WHERE
 NYC.trip_distance1 > NYC.trip_distance2 AND
 NYC.trip_fare1 < NYC.trip_fare2 WINDOW
AS ( SLIDE INTERVAL 'C'  ON COUNT 'D');
```

5.3 Experimental Setup

We have a total of 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 use Apache Storm (version 2.4.0) as a benchmark DSPS and deploy all join algorithms to this DSPS. We use Apache Kafka (version 3.5.0) for input data to the DSPS, while Redis is used as an in-memory database for distributed cache to maintain the state of the window for distributed parallel operators.

Table 1: Data set description

Data set	Record	Tuples for streams	Cite
New York Taxi	173M	R=173	[9]
BLOND	60M	R=30M, S=30M (6GB)	[14]
Synthetic	32M	R=32; R=16, S=16	

6 RESULTS AND DISCUSSION

In this subsection we provide experimental results for stream inequality join algorithms and delve into their implications. Initially, we provide a comparative results analysis for the SPO-join algorithm with state-of-the-art stream inequality join algorithms. Later, we discuss the impact of performance by employing different configurations for the proposed SPO-Join algorithm.

6.1 Mutable-Immutable Join structure

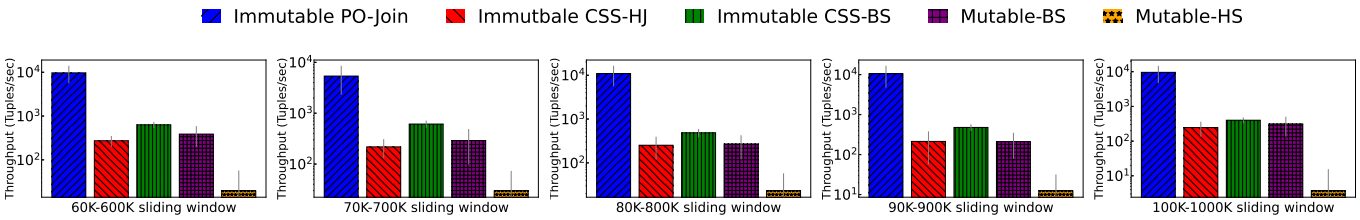
The algorithms that are used for inequality join conditions have two distinct structures. The SPO-Join algorithm and PIM-tree [23], are made up of a mutable B^+ tree and various immutable parts such as the PO-Join structure and CSS-tree. During the stream processing, after a certain interval of time or count, the mutable part is merged into the immutable part.

Experimental description: In order to conduct a comprehensive evaluation of both data structures, we have analyzed the throughput and event time latency of the SPO-Join algorithm and PIM-tree-based mutable and immutable parts for Q 1 and Q 2 with varying window sizes. For the self-join query, our slide interval ranges from 60K to 100K and the window size changes from 600k to 1M. For the two-way join, the slide interval varies from 100k to 500k, and the window size ranges from 1M to 5M tuples. In order to conduct the experiment for Q 2, we have used 7 nodes, whereas for Q 1, which is adopted for two-way join and intersection, we have deployed 10 nodes. During the experimentation, we observed the mutable part data computation, especially the intersection part, using both default hash-based and bitset-based evaluation. Similarly, for the immutable part, we have used an equal number of parallel operating tasks for fair evaluation. The CSS-based immutable part requires another level for intersection, so we have observed both the bitset and hash set for Q 2 and for Q 1, we have observed only the bitset due to its better performance. Moreover, for Q 1, we have adopted slide interval division instead of whole sliding interval for merging tuples from mutable components into immutable data structures for larger window sizes. For all experiments, the input insertion rate of tuples to the DSPS is the same.

Results: The graph shown in Figure 7 portrays the throughput for a taxi dataset with different sliding intervals and window sizes. The x-axis displays the scheme name, while the y-axis shows the mean and standard deviation of tuple processing throughput. Figure 7a highlights the results for a 60k slide interval and 600k window size. This graph demonstrates that the PO-Join algorithm is 35x and 16x times better than the hash-based and bit-based immutable CSS tree-based join algorithms, respectively. Additionally, for the mutable component, the graph shows that the bit-based stream join has 19x superior performance than the hash-based mutable component. 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 bitset-mutable join is 3.6x times better than the alternative.

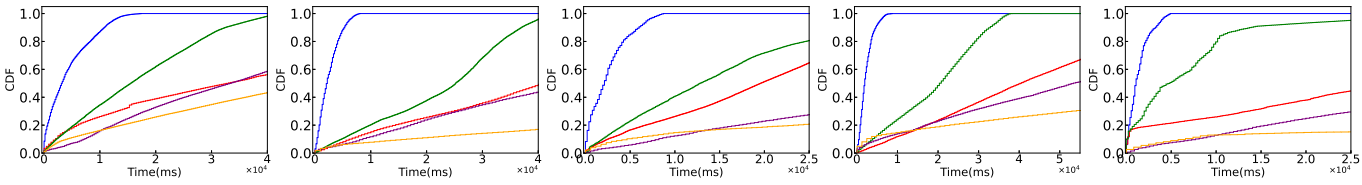
Similarly, Figure 7b shows the throughput for 70k and 700K sliding window. In this case, the proposed immutable component of the stream PO-join algorithm has 25x and 12x time superior to alternative immutable components. Similarly, a bit-based mutable join is 9 times better than hash-based join. Figure 7c depicts the self-join results for an 80k slide interval with an 800k window size. The results of the figure show the analogous behavior of improvement of the proposed SPO-Join algorithm, where the immutable PO-Join algorithm has 43x and 22x time better than the alternative. Moreover, the mutable part has 12x time better than hash-based join for mean value throughput. Similarly, Figure 7d and Figure 7e depict the results of higher slide intervals and window sizes. The result of Figure 7d depicts that the proposed PO-Join has 49x and 22x time better than CSS tree-based hash and bit join algorithms for 80k intervals and 800k window intervals. However, the mutable bit-based join algorithm has a 17.5x time better mean performance than the hash-based mutable join algorithm. Similarly, our observation for 100k slide interval depicts that 39x and 24x time better performance, however, in case of higher limit by standard deviation, the proposed immutable stream PO-Join has 57x and 31x time better performance than alternative immutable components. Similarly, the mutable component has 44x time better throughput for higher limit.

The event time latency for the Q 2 has been described in Figure 8. The cumulative distribution function of latency against 60k slide interval and 600k window size with 6 processing elements for immutable components is depicted in Figure 8a. The join process takes 3.2 seconds for the 50th percentile of tuples in the immutable part of the SPO-Join algorithm, whereas hash and bit-based immutable join consume 28 sec and 15 sec. In the mutable part, the proposed stream takes 33 seconds, while the hash-based join takes 48 seconds. For the 75th percentile, the proposed stream PO-Join takes up to 7 sec, whereas the other two categories consume 67 seconds and 25 seconds for the immutable part of the streaming window. For the mutable part, the hash-based algorithm takes 22 seconds more than the proposed bit-based join strategy. The 95th percentile also depicts similar behavior, where the proposed immutable component of stream join takes 12.4 seconds, while alternatives take 109 seconds and 37 seconds. Moreover, the mutable component is 17 seconds faster than the hash-based strategy. Similarly, Figure 8b depicts the cumulative latency for 70K and 700K slide interval and window size with 7 processing elements for each immutable operator. Results show that the immutable PO-Join latency has taken 2.4 seconds for the 50th percentile of tuples, whereas hash and bit-set-based immutable parts have 40 seconds and 25 seconds each. Similarly, the mutable part consumes 48 seconds and 140 seconds for the bit and hash-based B^+ tree mutable part. For the 75th percentile, the PO-Join-based immutable part consumes 3.824 seconds, whereas alternatives take 82 seconds and 31 seconds each. The mutable components consume 83 seconds and 249 seconds for tuple processing. The 95th percentile for PO-Join is 8 seconds, which is 10 times better than the CSS-tree-based bit-join strategy. Moreover, the bit-based mutable join is 2.2 times better in time consumption.



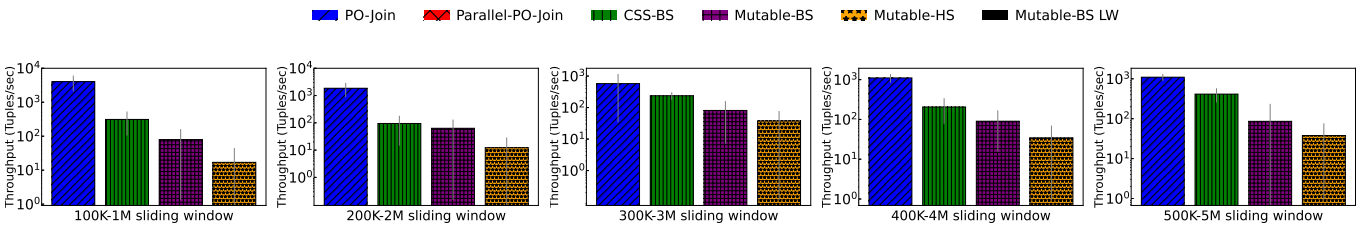
(a) 60K-600K (b) 70K-700K (c) 80K-800K (d) 90K-900K (e) 100K-1000K

Figure 7: Throughput for NYC self join Q 2 with varying W size



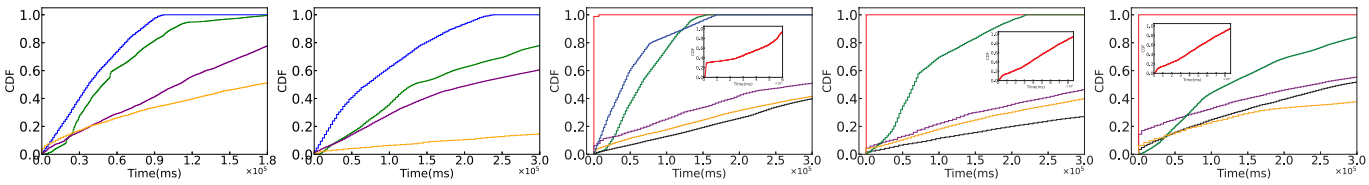
(a) 60K-600K (b) 70K-700K (c) 80K-800K (d) 90K-900K (e) 100K-1000K

Figure 8: Event time latency for NYC self join QQ 2 with varying W size



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

Figure 9: Throughput for BLOND data set for two-way join Q 1 with varying W size



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

Figure 10: Event time latency for BLOND data set for cross join Q 1 with varying W size

The commutative frequency distribution of latency for 80k and 800k slide interval and window size with 8 processing tasks each is shown in Figure 8c. The results indicate that the PO-Join algorithm takes less time than other alternatives. For instance, for the 50th percentile, PO-Join consumes 2.217 sec while the hash and bit-based CSS join algorithms require 19 sec and 12 sec, respectively. Additionally, the proposed mutable join algorithm performs 2x better than the hash-based B^+ tree join strategy. Similarly, for the 75th percentile, the proposed PO-Join-based immutable component only takes 4 sec, whereas the CSS tree with hash and bit join algorithm takes 29 sec and 21 sec each. The mutable component is usually 2x better than the alternative. The 95th percentile depicts that the immutable PO-Join strategy takes 7.2 sec, while alternatives consume 33 sec and 38 sec each. Moreover, the mutable component consumes 204 sec and 270 sec for bit and hash join strategies.

Figure 8d (d) depicts the CDF of latency against 90K and 900k slide interval and window size with 9 processing elements each immutable component. Results depict that the proposed immutable

component has consumed 2 sec for the 50th percentile of tuples, whereas other immutable algorithms, including hash and bit-based CSS algorithms, take 40 sec and 20 sec. Similarly, as analogous to the previous discussion for the mutable component, the proposed bit-based tree join algorithm has 2x times better performance than the alternative. Similarly, in the case of the 75th and 95th percentile, the proposed PO-Join-based immutable part and bit-based intersection for the mutable part of the B^+ tree have superior performance than the alternatives.

Figure 8e shows the same behavior for 100k and 1000K sliding intervals and window size for 10 processing elements. For the 50th percentile, the proposed immutable PO-Join-based scheme only consumes 1.1 sec, whereas other schemes consume 30 sec and 5 sec for each. However, with an increase to the 75th percentile, the proposed immutable stream PO-Join strategy consumes 1.9 sec, whereas others take 52 sec and 9.6 sec each. Similarly, superior behavior of performance is observed for the 95th percentile of streaming data.

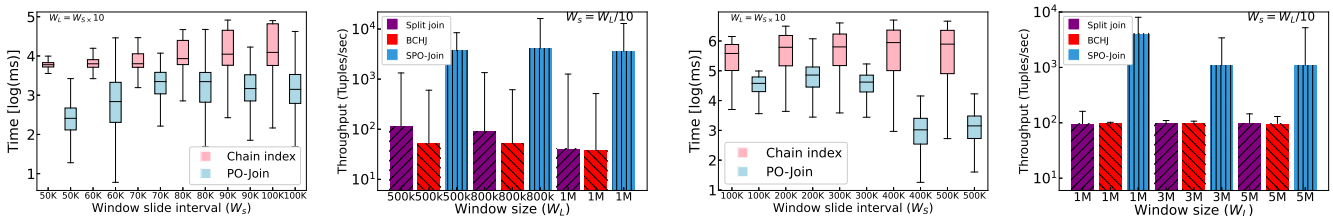
The results of stream inequality join algorithms for Q 1are depicted in Figure 9a to Figure 9a. In this case, we only considered an immutable CSS tree with a bit-set intersection due to its superior performance than a hash-based intersection. Figure 9a shows the throughput for Q2 with a 100k slide interval and a 1M window size. The results indicate that the proposed PO-Join algorithm has 12x higher throughput than the bit-based CSS tree immutable component. Similarly, the proposed bit-based join strategy performs 4.6x times better for the mutable component than the hash-based intersection of B^+ tree evaluations. Moreover, Figure 9b shows the results of a 200k slide interval with a 2M window. The results indicate that with an increase in tuples, the throughput decreases. However, the proposed immutable component has 19x better performance than the alternative. Similarly, the mutable component has a 5.25x better performance than the hash-based evaluation. Figure 9c depicts the throughput for a 300k slide interval with 3M tuples. In this case, the slide interval size is long, so we divide the slide interval into the available processing elements to minimize the merging cost. The results indicate that the proposed PO-Join structure has 2x better performance than the CSS tree-based join strategy Similarly, our bit-based mutable component also performs 2x times better than hash-based join.

Furthermore, Figure 9d depicts that the proposed PO-Join structure has 5.38x better performance than the alternative. Similarly, the mutable component has 2.6x better performance than the hash-based mutable structure. Figure 9e depicts that the proposed PO-Join strategy has 2.65x better performance than the immutable CSS join structure. Moreover, the mutable part has 2.297x better throughput than the hash-based strategy. However, the standard deviation of the mutable component for hash-based join is very high, and it almost approximates the mean for bit-based mean value.

Figure 10a(a) to figure 10b display the commutative frequency distribution of event time latency for Q1. The graphs showcase the performance of various methods used for joining the streams. These include an immutable stream PO-Join structure, an immutable CSS tree-based join algorithm with the bit-based intersection between opposite predicates, a mutable B^+ tree with the bit-based intersection, and a mutable B^+ tree with the hash-based intersection. 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. Moreover, the mutable part has twofold better performance than the alternative.

Figure 10b displays the distribution frequency of latency for a 200k sliding interval and 2M window size. The graph shows that the immutable component takes 71 seconds, whereas the CSS-based tree consumes 133 seconds. Moreover, the mutable component has 9.2x superior performance than the hash-based intersection solution. Similarly, the 75th percentile depicts that the immutable CSS-tree join algorithm takes 2x more time consumption than the PO-Join-based solution. Moreover, the bit-based mutable component has 7.2x superior performance than its alternative. The 95th percentile shows similar behavior where the proposed immutable data structure has 2.3x better performance than the alternative immutable design, whereas the mutable component has 11x better performance than the hash-based join algorithm.

Figure 10c shows the CDF plot for 300k and 3M window size. In this case, we experiment with the immutable component in two ways: (1) use the normal division of slide interval in a round-robin way, (2) divide the slide interval depending on the downstream distributed instances, and maintain a consistent state of window among all processing elements. Results depict that for the 50th percentile; the proposed divided slide interval PO-Join structure has 70x better than normal round-robin slide interval distribution. Similarly, for other strategies proposed immutable part outperforms in terms of time consumption. Moreover, for the mutable part, the divided sliding window with B^+tree has 1.5x time better in join computation than complete slide interval. For the 75th percentile, the divided slide interval has 44.6x for immutable and 1.5x for mutable parts superior to its counterpart (full slide interval). Similarly, improved performance is observed for the 95th percentile where the divided sliding interval-based approach has 7.3x for immutable and 1.3x for mutable components for the stream join algorithm. In Figure 10d and Figure 10e show the latency distribution for 400k-4M and 500k-5M streaming window. The immutable sliding window for the PO-Join structure has further parallelism inside an operator. Each processing task holds the linked list of slide intervals, where a new tuple performs a join operation parallel on every index of the linked list depending on the cores of hardware for the task. Figure 10d shows that for median tuple processing the proposed immutable component has 60x improved than the alternative. Similarly, for the 75th and 95th percentile, the immutable component has 67x and 36x better performance. Moreover, a mutable component with a divided slide interval has 1.7x, 1.17x, and 6.9x time better consumption for the 50th, 75th, and 95th percentile of tuple processing than the alternative. Similarly, Figure 10e depicts that the proposed immutable component with scale-up processing takes 233ms for the 50th percentile while CSS-based with sub slide



(a) Latency for Chained Index and PO-Join for self join against Q 2

(b) Throughput for BCHJ, SJ, and SPO-Join for self join against Q 2

(c) Latency for Chained Index and PO-Join for cross join against Q 1

(d) Throughput for BCHJ, SJ, and SPO-Join for cross join against Q 1

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

window interval takes 123sec, similarly, 75th and 95th percentile takes 475sec and 659 for PO-oin based stream join, however, alternative time consumption is 243sec and 422sec each. Moreover, Figure 10e(e) figure also depicts the superior performance of the bit-based mutable join algorithm with other mutable structures.

Analysis: Both designs offer two components for the whole stream in-equality 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 by and executed by the mutable component. The better performance of Q1 and Q2 for PO-Join-based immutable component w.r.t CSS tree structure is due to two reasons. 1) a new tuple in the stream PO-Join data structure is searched using a binary search algorithm on the consecutive memory location of sorted tuples in the window. However, in CSS-tree tuples exist on the leaf nodes of the tree and these tuples are packed using blocks and distinct nodes that are linked together. So the linear scan on linked nodes is much more expensive than consecutive memory location 2) The intersection procedure by stream PO-Join is less expensive as the data structure contains the memory location of streaming tuples that have constant cost to fill or scan the bit array. However, the CSS-based join structure waits for both streams to complete their predicate evaluation. Subsequently, they send their individual result to another reducer to intersect the parent predicate results. This procedure consumes more time to fully process a tuple; moreover, it requires an extra level of data provenance for correctness. The primary distinction in the mutable design lies in its utilization of bit sets and hash sets for intersection operations. In the case of a bit-based intersection, there is no need to calculate the hash function for each result of the resulting predicate. Instead, a single bit is allocated for each tuple, and the intersection is then computed through a straightforward bit-wise operation. 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 merges 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.

6.2 Chained Index Structure

Experimental description: In this experiment, we evaluate the Q1 and Q2 for proposed stream inequality join data structure against the chained index [16]. In this experiment, we consider only the archived sub-indexes of a chain of B+trees index structures and the immutable data structure of the proposed join algorithm. For Q1 we adopt 7 machines and for Q2 we use 10 machines with an equal number of worker processes (10) and parallel processing elements both for active or archived for chain index and mutable and immutable components for the proposed IE join structure. For the chained index, each new tuple is partitioned in two ways such as hash partitioning for insertion in the active sub-index and broadcasting tuples to the same downstream processing for predicate evaluation. Moreover, another level of operator performs intersection among upstream results of each individual predicate. We observe only the event time latency; which considers the tuple time from insertion to the DSPS system and then finally producing the result.

Results: Figure 11a and Figure 11c show the event time latency with varying slide interval and window sizes against Q1 and Q2. 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 against chain index. Moreover, for 75th percentile; the IE-join structure has 14.3x, 3.7x, 3.0x, 6.5x, 13.6x, 19x better than alternative. Similarly, for the 50th percentile the 23x, 9.19x, 3.2x, 3.8x, 7.5x, and 8.7x the proposed PO-Join structure has prime performance than chain index Figure 11c depicts the results for 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 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. Moreover, the 50th percentile of data also depicts the better performance of the proposed SPO-Join structure than the alternative.

Analysis: Chain index is based on the linked set of balanced search trees, where a new tuple can only be inserted to the last active sub-index from the linked data structure. One of the key disadvantages is that increasing the size of the sub-index increases the search cost with a small improvement in insertion (index only in sub-index). A tuple requires to search of every sub-index with different memory locations and without parallelism it increases the search procedure inside the processing tasks. This procedure also increases the waiting time for tuples in the queue. Meanwhile, the proposed PO-Join holds the sorted data items in a contiguous

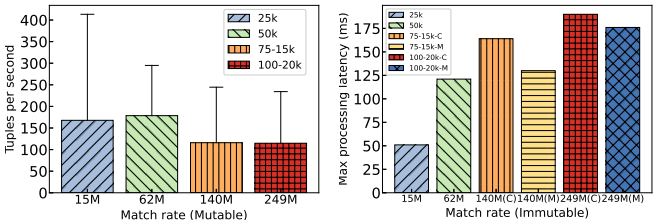
memory location. The linear scan on contiguous memory location is less expensive than the scan on the balanced search tree leaf nodes. Moreover, we subdivide the slide interval into sub-intervals for higher slide intervals and further parallelize the SPO-Join data structure search procedure on the downstream processing task node.

6.3 Partitioning based Stream Join

Experimental description: In this experiment we compare the performance of the proposed stream IE-join algorithm with broadcast hash join and split join algorithm for equality stream query processing. Both of these algorithms do not require a special data structure for the window as data items are held by the simple dynamic arrays. For broadcast hash join, a new tuple is broadcast to all downstream instances for insertion where a single hash function is also applied to select a downstream task for tuple predicate evaluation where simple loop join computes predicate result. Moreover, the final operator is also required for intersecting the results of upstream processing elements results. Similarly, split join used round-robin data partitioning for insertion and broadcast partitioning for tuple evaluation. For fair evaluation, we use an equal number of nodes and worker processes for these partitioning-based schemes and proposed SPO-Join structure. We choose throughput as a performance metric due to the difference in tuple processing strategy by the partitioning-based schemes and proposed data structure.

Results Figure 11b depicts the throughput results for stream SPO-Join with SJ and BCHJ algorithm against Q 2. 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. Moreover, with an increase in slide interval and window size, the performance of BCHJ and SJ is identical, however, the proposed data structure has 90x time better performance than the alternative. Figure 11d depicts the results of throughput for Q 1 two-way join with larger slide interval and window size. Results depict that the proposed join data structure has 43x superior performance than the alternative, moreover, it is also observed that slide intervals do not impact too much on partitioning schemes.

Analysis The partitioning-based schemes use a simple nested loop strategy to evaluate the processing tuples where each tuple is required to expedite the whole list for complete predicate evaluation, moreover, after both predicate evaluations, another operator is required to perform intersection operations. However, the proposed data structure uses a B^+ tree for the insertion of tuples and uses an efficient immutable structure that holds the window items in a sorted array with a contiguous memory location. Moreover, the immutable part does not require another level of operator to complete the whole predicate operation. The fast processing by the data structure also minimizes the waiting for a tuple in the queue. Another observation is that with an increase of the window size the performance of BCHJ and SJ is identical. The SJ is required to accommodate the results from all processing elements for complete join as BCHJ completes the whole single predicate operation from a single operator.



(a) Throughput for varying match rate for mutable part

(b) Latency for varying match rate for immutable part

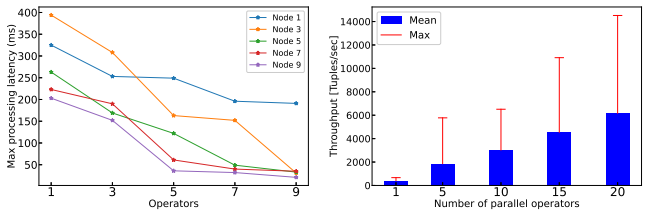
Figure 12: Impact of match rate for mutable and immutable structure

6.4 Match Rate

Experimental description: We use a synthesized data set with increasing match rate varying from 15M to 249M result set for slide interval. For each experiment 10 computing nodes, 9 worker processes, 7 processing elements for whole mutable data structure, and 5 tasks for these PO-Join based immutable data structure. For higher matching rates including 140M and 249M result sets, we use subdivision of slide interval for merging from mutable to immutable data structures. For mutable data structures, we aim to demonstrate how much the throughput deviates when we transition from a lower match rate to a higher match rate while using sub-slide intervals. In the case of immutable data structures, we consider processing latency as a key performance metric for our algorithm. We also present two designs for processing tuples within high-match rate slide intervals. These designs involve increasing the processing elements (scale out) and increasing the number of threads on single hardware that depends on the size of the immutable data structure and available cores.

Result: Figure 12a shows the average processing throughput and its variability from the mean value. It depicts that, for a 15M match rate the mean throughput is 167 tuples/sec whereas it varies up to 245 tuples/sec. Similarly, for a 62M match rate and 50k for slide interval, the mean throughput is 179 tuples/sec with a 2.11x change on its standard deviation than the 15M match rate. Moreover, for higher match rates 140M and 249M, the results show that the average throughput is 115 tuples per second and 114 tuples per second, respectively. However, the standard deviation for a slide interval of 15,000 is 128 tuples/sec, while for a 20,000 slide interval, it is 119 tuples per second. Figure 12b depicts the processing latency for an immutable component of the proposed data structure with varying match rates. It shows that maximum processing latency for 15M is 51ms for 25K of slide interval (each item in the linked list of immutable data structure has 25k of size), similarly, for 62M match rate and 50K slide interval the maximum processing latency is 121ms. For the higher match, the size of the linked list is increased; results depict that maximum processing latency with processing elements (scale-out) is 164ms and 190ms for 140M and 249M match rates. Similarly, processing with an increasing number of threads (scale-up) depicts 130ms and 176ms for a higher match rate.

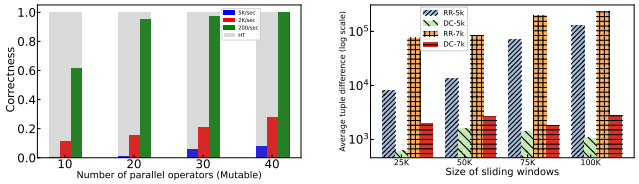
Analysis: In the mutable data structure, the sub-division of slide interval helps to improve the throughput for a higher match-rate



(a) Increasing number of machines

(b) Increasing number of operators

Figure 13: Scalability with increases resources



(a) Correctness for the mutable part, comparison with hash table

(b) Correctness for immutable part with & without cache

Figure 14: Correctness for mutable and immutable part

sliding window interval. However, a slightly slow performance of throughput for higher sub-divided slide intervals is due to more rapid merging than the alternative. Similarly, for immutable data structure; for full slide interval the maximum processing latency is 2x higher for 50k slide interval than 25K. However, after the introduction of sub-interval merging on processing elements, this raise decreases to only 1.3x and 1.5x with several processing elements and 1.07x and 1.4x for single processing element and multi-threading against 140M and 249M match rate slide interval window. The subdivision removes the overhead of processing a whole slide interval by a single processing element. Moreover, the use of multi-thread instead of more processing elements also has better performance, however, it is strongly constraints with available hardware, moreover, increases in the size of the match rate also increase the number of immutable data structures on downstream processing tasks, that also degrade the performance with multi-threading.

6.5 Scalablity

Experiment design In this experiment we test the scalability in two way; 1) increasing number of nodes; 2) increasing number of processing elements. We observe the maximum processing latency for varying nodes and mean and maximum throughput with increasing processing elements on the tasks. Here we utilize a synthesized dataset with a 15M match rate of tuples, moreover, machine size varies from 1 to 9 with fixed 5 processing tasks, similarly for processing task changes from 1 to 20 with 9 computing nodes in the cluster.

Result Figure 13a (a) shows the result of scalablity with an increasing number of nodes. The maximum processing latency on the first processing element for 1 node is 325ms, whereas on the 5th processing element, it is 191ms. Moreover, with node 3 these latencies are 394ms and 31ms. However, with an increase in processing nodes, the maximum processing latency on processing elements is decreasing. Finally, with 9 processing elements, these latencies are 203ms and 21ms. Similarly, Figure 13b depicts the result of throughput with an increasing number of processing elements. It can be seen that this mean throughput rises from 419 tuples/sec to 6167 tuples/sec from increasing 1 processing element to the 20 processing elements. Moreover, the maximum throughput increases from 668 tuples/sec to 14519 tuples/sec.

Analysis The increasing number of machines distributing the input tuple load to other machines definitely impacts the processing latency of each tuple. Similarly, increasing the number of processing elements reduces the impact of waiting time in the queue. For smaller processing elements, more slide intervals are added into the immutable data structure where new tuples expedite all intervals for processing. However, a larger number of processing elements distribute these mutable structures to all available tasks. A new tuple is broadcast to all processing tasks that can independently process each tuple in parallel, increasing processing tasks also reduces the waiting time tuple in the queue for larger window sizes.

6.6 Correctness

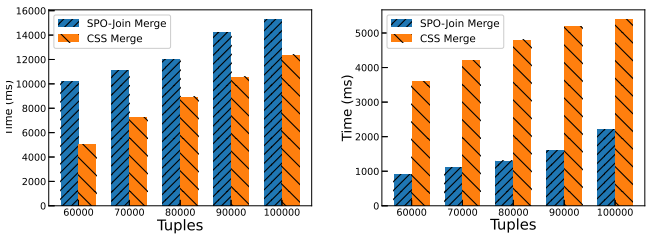
Experiment design We conduct this experiment separately for mutable and immutable structures. For mutable data structure, we observe the correctness of the intersection operator for join data structure with different insertion rates of tuples. We measure the impact with an increasing number of parallel processing elements and insertion rate with a 25k slide interval window for hash-table-based and without hash-table-based joining. Similarly, for immutable data structure, we measure the correctness with different state management strategies for sub-slide intervals with high insertion rates. In this case, the size of the downstream processing element is 5 where each immutable data structure contains slide-interval/|processing-elements| tuples for each processing task.

Results: Figure 14a depicts the correctly computed join results for the input tuple. Results depict that for higher insertion rates such as 5000 tuples/sec and 10 downstream processing tasks, only 0.3% tuples are correctly computed in their join results. However, it shows that with an increase of processing elements, the percentage of correctly identified tuples is also increasing. Meanwhile, the high insertion rate does not allow it to ensure 100% correctness as depicted for 40 processing tasks and 500 tuples/sec; this correctness approaches only 8%. However, partitioning and use of a small hash-table data structure guarantees 100% correctness for all input tuples. Similarly, smaller insertions; such as 2000 tuples/sec and 200 tuples/sec ensure slightly better correctness with increasing processing elements as shown that for 200 tuples/sec and 40 processing tasks, the correctness approaches 98%. Figure 14b shows the tuple differences between different processing elements on the cluster for both of our proposed window state management strategies for sub-sliding intervals against high input data rates. For a 25k slide interval and 5000 tuples/sec, the average difference between the tuples processing count between the first processing elements to others is 13x greater for distributed cache-based tuples state synchronization than for the round-robin scheme. Similarly, for 7000 tuples/sec, this difference is 38 times better than the alternative.

Moreover, for higher slide intervals and high insertion rates, this difference is 82x and 94x that depict the use of distributed cache among processing instances reduces the false positive rate of tuple processing for a sliding window.

Analysis: For mutable data structure; the result of the new tuple is downstream toward an intersection operation after both predicate evaluations using hash partitioning. However, in the case of high input data rate and different sizes of data structures on each predicate operator, there are differences in tuple processing tuples that ultimately impact on correctness. However, the use of a small hash table holds the tuple with <id, tuple> in its table data structure, where it only performs intersection operation depending on the match of the key from both predicates. It only increases a small overhead of memory and removal of tuples from the data after join completion. Similarly, for an immutable data structure, we are assuming that the first processing element contains the actual state of the window (count). With input data rate more tuples are processed by these processing elements. However, for the round-robin strategy, we only use the size of mutable data structure as window size for all other processing elements except the first processing task which raises the difference between window size synchronization among all tasks. Similarly, a distributed cache-based strategy decreases this difference as other processing elements synchronize their window state with the first processing elements after a predefined interval or synchronization. Note interval definition is subjective to the use case, smaller intervals have high correctness but raise a bit overhead of high state synchronization.

Figure 15a shows the merging overhead of both Q1 against with NYC data set (experimental setup is same as for a self-join). Results depict that the proposed SPO-Join structure consumes 10sec to 15sec for merging the tuples from 60K to 100k slide-interval windows. However, CSS-tree-based merging has consumed 5sec to 12sec for the same slide-intervals. The SPO-Join structure computes an extra permutation array that raises some overhead of extra computation. However, the CSS tree also requires building a new data structure for immutable structure in downstream processing tasks. Results depict that for 100k processing tuples, CSS-tree only performs 1.2x time better than SPO-based merging. Similarly, Figure 15b depicts the merge tuples evaluation; where it can be seen that for proposed IE-join data structure has more than 2.5x greater than the alternative for 100K tuples.



(a) Merging time **(b) Merging tuple evaluation**

Figure 15: Impact of merging for IEJoin based stream inequality join

7 RELATED WORK

In this section, we explore the existing literature and shed light on many algorithms and techniques used for stream join.

In-memory join Growing need for real-time data processing, queries now necessitate retaining intermediate processing results in the system's memory instead of I/O operation [1]. This strategy is employed to expedite the search process. A chain of research work has been investigated that considers the in-memory query processing [2–4, 8, 15, 19, 21, 28]. These techniques mostly provide different optimization for paralleling the nested loop join, redix sort, and hash join approaches for distributed parallel or multi-core systems. However, our solution is only considering the window-based stream inequality join for distributed stream processing systems (DSPSs). Moreover, existing methodologies consider the optimization with in-memory join for fixed data from databases. Moreover, the solutions discussed primarily emphasize equi-join queries, where the join condition involves equality comparisons. However, these approaches can become costly, exhibiting quadratic time complexity, when applied to non-equi join queries.

Sliding window-based stream join Verwiebe et al. [27] provide a comprehensive review on different windowing strategies used to bound the streaming data contents, however, many studies considered only the sliding window for stream join operation where a single tuple can belong to more than one window and perform join predicate results for more than one window [6, 7, 13, 16, 17, 20, 23, 25, 31]. In handshake join [25] both streaming data flow in the opposite direction where the predicate evaluation is only carried out during stream by. Low latency handshake [20] (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 single tuple. Split join [17] 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. A sliding window is divided into several sub-windows depending on the number of cores or processing elements in distributed settings. A new tuple is inserted into the dedicated core depending on the round-robin approach, where as the opposite stream tuple is forwarded to all cores to produce the predicate results. Similarly, chain index [16] and PIM tree [23] 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 immutable components to speed up the tuple evaluation process. Furthermore, the current strategy lacks a clear methodology for maintaining the state across processing cores or updating sliding windows. The proposed solution for stream inequality joins goes beyond singular predicate evaluation; instead, it comprehensively addresses multiple predicates and incorporates intersection operations among them.

Stream join in distributed stream processing system A number of studies target distributed stream join processig [4, 10, 11, 16, 24, 26, 29, 30, 32]. Bistream [16] 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, moreover, no proper state

management strategy is discussed among the processing element that holds these chain indexes. AJoin [11] is a two-layered architecture that includes optimization and stream processing layers. The optimization layer periodically optimizes the execution plan whereas the processing performs the stream join processing in an incremental way. FastJoin [32] considers the problem of load imbalance among the processing elements of DSPS under the presence of skew in the input data. A GreedyFit methodology is adopted to filter out the key that can be the potential causes of load imbalance and propose an online migration strategy of tuples among processing elements to the load imbalance among processing elements. Our approach employs the round-robin strategy to distribute PO-Join data to downstream instances, ensuring a balanced distribution among PO-Join operator instances.

8 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 which outperforms traditional indexing strategies. We propose a novel approach that exploits both a mutable and an immutable index. New tuples are inserted into the mutable B^+-tree (good for fast insertion), which is periodically merged into an immutable PO-Join structure (good for fast search). The majority of tuples in the sliding window are maintained by the PO-Join structure, while only new tuples are added to the mutable part. We also propose a novel algorithm to efficiently build the PO-Join structure in the streaming scenario. Finally, we conducted extensive experiments to evaluate the performance of our proposed solution. The results show that our approach outperforms other index-based solutions, making it a promising solution for stream inequality join queries.

ACKNOWLEDGMENTS

This work was supported by the [...] Research Fund of [...] (Number [...]). Additional funding was provided by [...] and [...]. We also thank [...] for contributing [...].

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proc. VLDB Endow.* 5, 10 (2012), 1064–1075.

[2] 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 *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[3] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 168–180.

[4] 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.

[5] 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 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[6] 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.

[7] 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.

[8] Bingsheng He and Qiong Luo. 2006. Cache-oblivious nested-loop joins. In *Proceedings of the 15th ACM international conference on Information and knowledge management*. 718–727.

[9] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 266–268.

[10] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 1507–1518.

[11] 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.

[12] 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.

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

[14] 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.

[15] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

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

[17] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. {SplitJoin}: a scalable, low-latency stream join architecture with adjustable ordering precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 493–505.

[18] David A Patterson and John L Hennessy. 2004. Computer organization and design: The hardware/software. *Computer Organization and Design, The Hardware/Software Interface Morgan Kaufmann* (2004).

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

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

[21] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proc. VLDB Endow.* 16, 7 (2023), 1749–1762.

[22] 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*. 2523–2537.

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

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

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

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

[27] 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.

[28] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. 2023. Nested Loops Revisited Again. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3708–3717. https://doi.org/10.1109/ICDE55515.2023.00299

[29] Jianye Yang, Wenjie Zhang, Xiang Wang, Ying Zhang, and Xuemin Lin. 2020. Distributed streaming set similarity join. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 565–576.

[30] 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.

[31] 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 2021 International Conference on Management of Data*. 2089–2101.

[32] Shunjie Zhou, Fan Zhang, Hanhua Chen, Hai Jin, and Bing Bing Zhou. 2019. FastJoin: A Skewness-Aware Distributed Stream Join System. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1042–1052.