Stream-aware Indexing for Distributed Inequality Join Processing

Adeel Aslam[a], Giovanni Simonini[a], Luca Gagliardelli[a], Luca Zecchini[a], Sonia Bergamaschi[a]

[a] *University of Modena and Reggio Emilia, Modena, Italy*

Abstract

Inequality join is an operator to join data on inequality conditions and it is a fundamental building block for applications. While there exist methods and optimizations for efficient inequality join in batch processing, little attention has been given to its streaming version, in particular to large scale data-intensive applications that run on *Distributed Stream Processing Systems* (DSPSs). Yet, to design an inequality join in streaming and distributed setting is not an easy task: *(i)* indexes have to be employed to efficiently support inequality-based comparisons, but the continuous stream of data imposes continuous insertions, updates, and deletions of elements in the indexes—hence a huge overhead for the DSPSs; *(ii)* oftentimes real data is skewed, which makes indexing even more challenging.

To address these challenges, we propose the *STream-Aware inequality join* (STA), an indexing method that can reduce redundancy and index update overhead. STA builds a separate in-memory index structure for hotkeys, i.e., the most frequently used keys, which are automatically identified with an efficient data sketch. On the other hand, the cold keys are treated using a linked set of index structures. In this way, STA avoids many superfluous index updates for frequent items. Finally, we implement five state-of-the-art inequality join solutions for a widely employed DSPS (Apache Storm) and compare their performance with our STA solution both on two real-world data sets and a synthetic one. Results reveal that our stream-aware solution

Email addresses: `adeel.aslam@unimore.it` (Adeel Aslam), `simonini@unimore.it` (Giovanni Simonini), `luca.gagliardelli@unimore.it` (Luca Gagliardelli), `luca.zecchini@unimore.it` (Luca Zecchini), `sonia.bergamaschi@unimore.it` (Sonia Bergamaschi)

outperforms others.

Keywords:
Distributed Stream Processing System, Inequality Join, B^+tree Indexing, Augmented Sketch, Skewed Data Distribution

1. Introduction

Data streams are ubiquitous in the Big Data era and are attaining significant attention from the industries and the scientific community. Streaming applications include, for instance, real-time analysis of social networks [1], smart grids management [2], fraud detection [3], network intrusion identification [4], and online financial trend indicators [5, 6, 7]. All of these applications require meaningful insight with a continuous flow of data with low latency and high record processing throughput.

Distributed Stream Processing Systems (DSPSs) are evolving to provide real-time analysis of continuous data. Many enterprises deploy their services over these DSPSs, e.g., the leading Chinese taxi app DiDi[1] uses Apache Flink[2] for real-time analysis [8], Twitter[3] deploys Apache Storm[4] for many applications, such as tweet analysis, searching, and revenue optimization [9], and Freeman Lab at Howard Hughes Medical Institute[5] uses Spark Streaming[6] for real-time analysis of human brain actions [10]. A DSPS (such as Storm, Flink and Spark Streaming) deploys the application on a cluster of computers that perform the same business logic in parallel on distributed nodes on continuously-updating data. In particular, the DSPS represents the application with a *Directed Acyclic Graph* (DAG) where the vertices represent the operations (e.g., join or aggregation) and the edges show the direction of the data flow between vertices—abstracting the logic of the program to its parallel execution.

Stream join is a fundamental operation used in several applications such as recommendation systems, online car ride and sharing applications, online

[1]https://blog.didiyun.com/index.php/2018/12/05/realtime-compute/
[2]https://flink.apache.org
[3]https://twitter.com/
[4]https://storm.apache.org
[5]https://jeremyfreeman.net/
[6]https://spark.apache.org

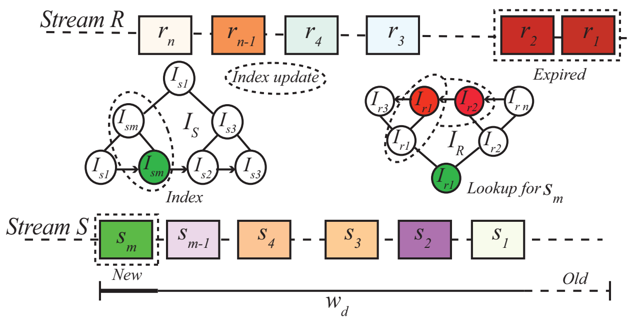

Figure 1: Window-based stream inequality join

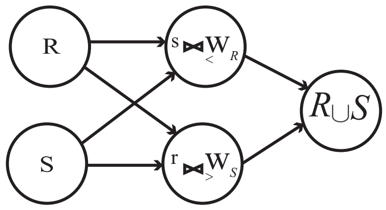

Figure 2: DAG for join processing of Query 1 (condition "R.V1 > S.V2")

monitoring and analysis of multi-sensor data [1, 11, 12, 13]. These applications provide analysis by joining the various streams of data based on certain conditions. However, it is impossible to save those infinite streams in the main memory to perform the join. Thus, the execution is bounded by a fixed amount of data that depends on a discrete-time interval or a number of tuples (i.e, a window). Join performed in this way is called *stream window join*, *window-based stream join* [3, 6, 14], or *windowing* for short in this paper when evident from the context. When the join requires to perform also inequality comparisons, such as "<", ">", "≤", "≥", it is called *inequality join*. This latter kind of join is widely employed and challenging, as we discuss in this paper. But first, let us consider the following example to get an intuition of the usefulness of this operator in a real use case.

Example 1.1. *Ellen is a data engineer for a fossil fuel power plant. She needs to monitor the performance of power generation during the conversion of mechanical energy to electrical energy. To ensure efficient and safe power-generating operations, Ellen looks for voltage (V) and current (I) readings in two given points of generators. These readings are performed by sensors installed on the input and output of the generators as well as on the transmission lines. Ellen writes Query 1 to monitor the generators, as explained in the following. R and S are two streams of individual V and I readings performed by two sensors of the generators, respectively. The two inequality conditions denote that the query will return the readings where the input V1 is greater than the output V2, and the input I1 is less than the output I2 for a time window of 10 seconds. This supports the monitoring of fluctuations and abnormalities during the power generation process. For example, in the case of voltage drops or current spikes, an investigation can be performed to* *prevent damages or power outages.*

Query 1: Query for real-time analysis of voltage and current stream with continuous data.

```
SELECT *
FROM R JOIN S
    ON R.V1 > S.V2 AND R.I1 < S.I2
WINDOW w AS (RANGE INTERVAL '10' SECOND PRECEDING)
```

To fully evaluate the stream inequality join of the example with a single predicate, two sliding windows are required. In particular, consider the example of the predicate "R.V1 > S.V2" from Query 1. This predicate implies that the voltage at the input of generator R, represented by the variable V1, is greater than the voltage at the output of generator S, represented by the variable V2 (i.e., the voltage at the output of generator S is lower than the voltage at the input of generator R), over a certain period of time. It becomes necessary to maintain two in-memory sliding windows, one for the stream R and the other for stream S, to evaluate the predicate for newly arriving tuples. The window-based stream join process is illustrated in Figure 1, while Figure 2 illustrates the DAG for processing the "R.V1 > S.V2" query predicate in a DSPS. In the DAG, each node represents an operator, such as a source operator that pulls data from a source (in the case of the example above, sensor data from input generator R or output generator S) or a join predicate evaluation operator (e.g., r>w_S and s<w_R, where r and s are single tuples from stream R or S and w_R and w_S are sliding windows for stream R or S. Moreover, a node with $R \cup S$ is used for aggregating results from the upstream windowing nodes). The predicate operators also maintain the sliding window and perform the evaluation on newly arriving tuples.

Hashing the tuples of the streaming data provides an efficient solution for search and update for equality join [3, 15]. However, for non-equality predicates, hashing is an inefficient solution as it requires too many memory scans for completeness. As for traditional batch DBMSs, indexing is a better choice to maintain the logical order of the contents [3, 16]. Nevertheless, the nature of streaming data differs from the one of traditional batch data. In stream join processing, records are continually added or deleted from the sliding window. Unlike traditional query processing, search queries, update, and delete are frequent operations for online data which create an index restructuring overhead for highly dynamic data as depicted in Figure 1, where I_R and I_S represent the index data structures for sliding windows of stream

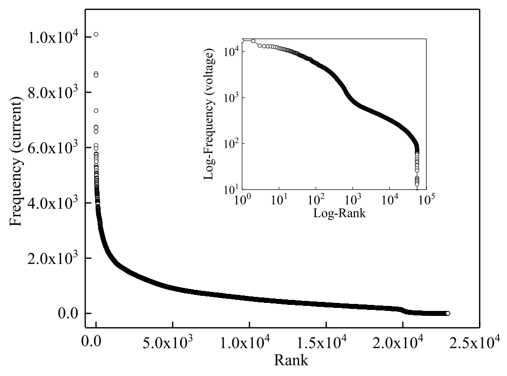

Figure 3: Frequency distribution of current and voltage data for BLOND dataset

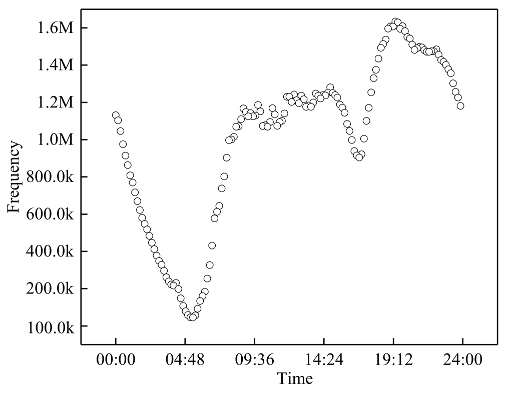

Figure 4: Frequency distribution of NYC taxi data w.r.t time

R and S. A new tuple s_m from stream S looks at I_R for the predicate evaluation and is indexed into I_S for future tuple evaluations. Moreover, the dotted circle denotes the change in index structure during insertion (*new*) or deletion (*expired*) of tuples from sliding windows.

The volatility of the streaming data prevents from attaining the full benefits of indices as in traditional databases. A single update in the sliding window may change the whole tree indexing structure. To overcome such an issue, several studies propose novel indexing strategies such as chain indexing [17], partitioned in-memory tree [3], Parallel-MJ [7], Panjoin [18], split-join [19], low latency handshake join [20], cell join [21]. The main intuition of these schemes is to use sub-indexes for a single sliding window that exist independently on each core of a single processor or worker process of a DSPS. These solutions eliminate the update and deletion cost to an extent, but at the price of creating an extra overhead for tuple accumulation or state management from several nodes or cores. Moreover, skewness in the streaming data also introduces extra costs for index updates and index redundancy on sub-indexes of the streaming window.

1.1. Empirical observation

In streaming, data skewness normally exists where some keys are more frequent than the alternatives. For example, mobile transportation [22], IP traces [23], click-stream data set [24], sensor readings, email, and SMS logs all follow the Zipfian distribution [25]. In particular, Figure 3 shows the data distribution of current and voltage for building level office environment data

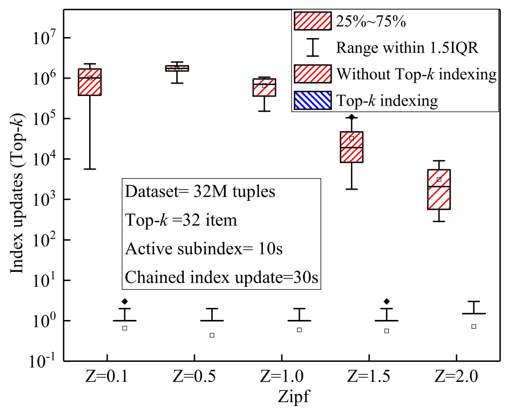

Figure 5: Index update for with and without top-k elements

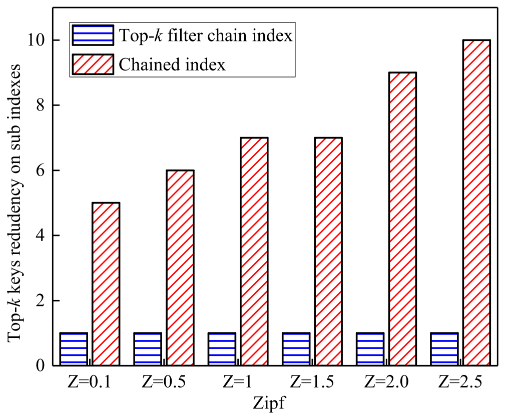

Figure 6: Redundancy of keys in sub-indexes

set [26], and Figure 4 shows the frequency distribution of taxi requests for New York City (NYC) taxi data set [27].

The proactive identification of the frequent keys from the streaming data and the use of a separate sub-indexing data structure for hotkeys can eliminate the update and deletion cost. Similarly, a separate sub-index for frequent keys also reduces the rate of redundancy on other sub-indexes. We also empirically analyze the update and rate of redundancy against synthetically generated varying Zipf distribution. Figure 5 shows the index update results for varying Zipf. Results depict that using a separate indexing data structure for top-k elements reduces the index updates on other sub-indexes. Similarly, Figure 6 shows that the use of separate indexing reduces the redundancy of top-k elements in the sub-indexes of the streaming window.

Several approaches have been proposed for stream data approximation includeing hash-based or key-based schemes: hash-based schemes such as counting Bloom filter [28] and count-min sketch (CMS) [29] use hash functions to keep track of items frequency. Yet, these schemes require a huge number of hash functions for a lower false positive rate. Another key-based strategy, Space Saving [30], also estimates the top-k items in the streaming data by maintaining a stream summary for hot items, where every key (hot or cold) enters into the stream summary. Unfortunately, these frequent exchanges among items may degrade the accuracy of hot items [31].

1.2. Contribution

In this paper, we propose to exploit *augmented sketch* (ASketch) [23] for the item frequency estimation in the context of streaming inequality join. ASketch is a two-layered approach with a small-size pre-filtering that filters the hot items and forwards all other tuples to the CMS sketch for approximation. One of the most significant advantages of ASketch is represented by the dynamical tracking of the popularity of the hotkeys in the stream, leading to better accuracy results with less space overhead. We use the results of ASketch for finding the hotkeys in the stream, then exploit separate index trees (I_h) for those frequent items. Moreover, all cold items use a chained-index of B$^+$tree (I_c) for the streaming window. In I_h, a fine-grained tuple deletion methodology is used where a new tuple can be added or deleted from the indexing structure depending on the item's popularity in ASketch. Furthermore, in I_c we adopt a coarse-grained tuple deletion methodology (as for the BiStream [17]) where a whole sub-tree is removed from the chain depending on sliding window time or count.

Besides indexing, data partitioning among processing instances of the DSPSs plays an important role in system performance. We use the power-of-two-choices (POTC) [32] for indexing hotkeys where hot items build their indexes among two processing tasks. This process yields a reduction in the load imbalance among processing elements compared to using a single choice [32]. Similarly, for all other stream items, a singly linked B$^+$tree is used among all processing elements of DSPS that maintain a shared state of indexed data structure. The processing element for a single item is selected randomly from the available set of tasks. An individual indexing data structure for every worker process creates a huge overhead of tuple accumulation among them. The use of a shared state among processing instances reduces this accumulation cost; however, it creates a small overhead of checkpoint interval where worker processes share the data structure state. We empirically analyze the impact of checkpoint interval for stream in Section 7 and show that higher skewed data has a lower tuple aggregation overhead among processing elements.

The overall contributions of this work are:

- We propose a novel indexing data structure for stream windowing items. The proposed solution keeps track of hot items in the streaming data and builds a separate indexing tree for all these keys. Moreover, all other items are treated by a chain of index trees.

- We devise a new data partitioning strategy that behaves differently for hot and cold items. All hot items are scheduled to fix two processing elements from a set of these instances to reduce the load imbalance among the worker processes of the DSPSs. Similarly, cold items are forwarded randomly to the processing elements except for the hot key instances.

- We implement state-of-the-art stream join solutions on the top of Apache Storm [9] and provide a thorough experimental comparison against real-world and synthetic data sets. The code is available at the following link [7].

This paper proceeds in such a way. Section 2 provides a thorough description of existing work for stream window join algorithms and data partitioning schemes. Section 3 discusses the preliminaries for the proposed work. Section 4 provides a description of the proposed stream-aware solution. Section 5 describes thorough implementation details of inequality join algorithms for DSPSs. In Section 6 we discuss the experimental setup for the evaluation, while Section 7 gives insight into the experimental evaluation and provides discussion. Finally, Section 8 concludes this paper.

2. Related Work

We classify the stream join related work into three categories: stream window join, state-of-the-art top-k prediction techniques, and data distribution among worker processes of DSPSs.

2.1. Stream window join

Kang et al. [33] provide three generic steps for stream window join. The stream join result provides all tuples $\langle r, s\rangle$ from the streaming window of R and S that satisfy the predicate condition. Moreover, a new tuple is inserted on its respective window, and expired tuples are deleted from the stream windows. This strategy uses the nested loop join for scanning the record of the window. However, in-memory structures such as a hash table for equality join predicate and index-tree structure for range queries provide better alternatives.

[7]https://github.com/AdeelAslamUnimore/StreamInequalityJoinSTA

A study by Teubner et al. [34] provides a solution for such hardware that utilizes a high level of parallelism for stream window join computation. This work is inspired by a soccer game where players from both teams shake hands with each other in opposing directions before the start of the match. Each tuple pushes the existing tuples of the window in a forward direction such that the oldest tuples always exist at the end of the window and expire. When $r \in R$ and $s \in S$ encounter each other, they evaluate the predicate like the soccer player handshake and evaluation result $\langle r, s \rangle$ is added into the result set. Roy et al. [20] analyze the latency for handshake join and this study experimental result depicts that the latency is strongly bounded by the size of the window.

Roy et al. [20] propose a low-latency handshake join (LLHJ) where the contents of the streaming window expedite from all available cores of the system. As analogous to handshake join, tuples of individual stream flow in opposite directions. Each core has local storage for holding the contents of streaming data. The core for tuple r or s is selected in a round-robin fashion. To speed up the search process, items on each core are maintained using B$^+$tree for local indexing. For predicate evaluation, a tuple $r \in R$ expedites from all cores for result $\langle r, s \rangle$ and then is added into the result set. Tuples from an opposite stream can be evaluated on the flight that can cause race conditions among threads [19]. Moreover, the tuples are deleted individually from each core that reshuffle the indexing structure of core c_i. Single tuple removal decreases the processing throughput and increases the concurrency overhead.

Split join [19] is a top-down data flow instead of two opposite streams. A single processing core has two regions (right and left): for stream window R and S. For example, a new tuple $r \in R$ goes to both parts of the core c_i. In the right part of the c_i , tuple r is stored and in the left the tuple predicate is evaluated against the stored tuples of stream S. A central coordinator is introduced that manages the parallelism among the cores of the CPU. However, the assignment of tuples to the joined for storage follows round robin procedure. Each join core locally manages the expiration of tuples depending on the time or count based. This study uses a simple nested loop for evaluating the predicates however, it is stated that hash join, or B$^+$tree can also be used. Split Join [19] has a single flow for both streaming data and depends on the number of cores. An increasing number of cores increase the split windows and this procedure increases the overhead of tuple accumulation from many cores. Moreover, individual tuple deletion from the single-core window creates extra overhead for concurrency control for expired tuples.

Lin et al. [17] introduce a novel model for stream join processing in DSPS named as Bi-Stream. This study claims that using a single index data structure for stream window items produces the extra overhead of index maintenance due to stream dynamics. To overcome this issue Bi-Stream introduces the chained in-memory index data structure that can hold the stream data items. It creates multiple index tree structures according to the tuple arrival time. All sub-indexes are linked as a linked list and each sub-index structure is archived as P. If the difference between the minimum and maximum time stamp for a tuple is more than P then this sub-index is archived, and a new sub-index is created. New tuples can only be inserted into the active sub-index, however, archived sub-indexes can be used for lookup. When the maximum time stamp of the archived sub-index is larger than a certain threshold, the whole sub-index is removed from the memory and released it for new incoming data. However, with the increased number of chained indexes search requires expediting the number of chains in the distributed nodes. This may increase the latency for tuple processing, especially for high-selectivity queries.

A study by Shahvarani et al. [3] propose a new design for indexing stream items of the window that compose of mutable and immutable data structure. The mutable data structure is the same as discrete B^+tree and it is inserted efficiently, whereas the immutable data structure is search efficient. The proposed indexing structure is better suited for high-selectivity queries. In mutable B^+tree each node consists of a pointer for the next node along the data pointer. Similarly, the immutable component is search efficient where multiple threads can read the data without race conditions. However, updates in the immutable parts are only applied during merger operations between two distinct trees. For range query, it is necessary to search both trees T_1 and T_S. However, when the tuple is expired it is marked and removed from the index structure during the merging of both T_1 and T_S. The individual range of the T_S indexing tree is defined by the insertion depth of the T_1. High-level insertion depth increases the number of linked B^+tree which increases the concurrent threads. A low level increases the height of B^+tree at T_S level, which creates the overhead for large reconstruction for new tuples.

A study by Pan et al. [18] propose a novel architecture for stream join processing. This architecture consists of a manager and a set of worker nodes. The streaming window is decomposed into several sub-windows on worker nodes. The manager receives the input and pre-processes the data and distributes the tuples toward the processing node for predicate evaluation. In order to speed up query processing three different data structures are introduced such as; a range partition table, wide B$^+$tree, and buffered index-sort. Range partitioning is used for small selectivity queries, however, for high selectivity queries BI-sort and wide B$^+$tree is utilized.

Khayyat et al. [35] propose a space-efficient inequality join solution (IEJoin) based on a bit array scan. For optimization of the bit array scan, a Bloom filter is introduced for fast computation of indices. IEJoin has two phases, *(i)* initialization, and *(II)* lookup. The initialization process includes first sorting of tuples based on index key and then permutation array and offset array computation that are quadratic in time complexity. However, the lookup phase is more efficient that only includes a linear scan of the bit array for the final computation of results. This work is well suited for fixed data, however, for streaming data, it is extremely expensive to maintain the computation array and bit array—hence, infeasible to achieve low latency in practice.

2.2. Stream data prediction

It is impractical to save and then manage the huge volume of continues data. Synopsis plays an important role for analysis and prediction of continues data. Cormode et al. [36] provide a comprehensive description about synopsis construction that include: sampling (random sampling, stratified sampling, chain sampling, priority sampling, and etc) [37], wavelets [38], sketches (count-bloom filter, count-min sketch, and cold-filter) [28, 29, 31], and top-k estimators (space-saving) [25]. Skecthes are normally used for to approximate the frequency of data stream element. Several studies considers these sketches for top-k identification of streaming data [39, 36], however, the top-k estimator (e.g, space-saving [36]) provide more accurate and efficient top-k return with high throughput [23]. Sketches need a large number of hash functions and high sketch size for higher accuracy, whereas, in space-saving algorithm their is a frequent exchange of stream that can degrade top-k prediction accuracy.

Several studies proposes a novel solutions by augmenting these state-of-the-art filter, sketches, and counter-based approaches (e.g., cold filter [31], stair sketch [40], loglog filter [41], augmented sketch [23]). The main theme behind these approaches is to separate the hot and cold items from the continues data. Cold filter [31] consists of three layers, where first two layers are small size filters that capture the cold items of the stream. Similarly, bottom layer has composed of sketch or counter-based algorithm for hot items. Stair sketch [40] mainly focus on the recent event stream and its accuracy instead of all items in data stream. Stair sketch considers the time stamps and organize the memory structure accordingly. It uses the basic bloom filter and CMS for testing the approach. Loglog filter [41] complements the cold filer [31] and filters the colds items of the stream where filter uses less memory space and hold more items than cold filter. Augmented Sketch [23] filters hot items of the stream by using two layer data structure. The first layer keeps key-value pair for item frequency whereas, second layer uses the CMS to approximation. Moreover, augmented sketch dynamically handles the popularity of the keys by swapping the items between layers depends on their frequency. In this work we use augmented sketch to keep track of hot item in the streaming element.

2.3. Data distribution among worker processes of DSPS

In DSPS the processing is performed on multi-node cluster. Number of worker processes run on the distributed nodes to handle the streaming data. Numerous data partitioning strategies have been used for data routing among processing elements of distributed node. These partitioning strategies selection depends on the data and business logic. The shuffle grouping [42] uses round robin scheme for partitioning the data among distributed nodes. It is better suited for stateless operation, however, in stateful data processing, data item requires an extra aggregation cost. Key grouping [42] uses the hash function for data distribution among processing elements. However, for higher skewed data it generates load imbalance among instances of DSPS due to the frequent key. Partial key grouping [32] uses two hash functions for single data item to reduces the impact of load imbalance among processing instances.

Identification of hot keys in a stream and partitioned them in more than two processing elements can reduces the load imbalance among instances of the data element. Nasir et al. [43] propose a hotkey-aware data partitioning solution. They use counter base space-saving data structure to keep track for top-k elements and route these elements to more than two processing elements using w-choices and d-choices where as cold items are treated by using power-of-two-choices. Chen et al. [44] propose a novel counting-based probabilistic solution for dynamic tracking of hot keys. All hot keys are processed by all worker processes in a round-robin fashion where cold items use

Table 1: Notation and there explanation

Symbols	**Explanation**
R,S	Stream R, Stream S
r,s	Tuples from stream r and s
W_R,W_S	Sliding window for stream R or stream S
d	Time unit
k_i, k_j	k_i insertion key and k_j search key
∂	Threshold for time-based sliding window
$\bowtie_\theta$	Predicate to evaluate
I_r,I_s	Indexing structure for W_R or W_S
h_b	Height of I_r, I_s
t_{di}^{w}	Expire tuple
t_{kr}^{max}, t_{kr}^{min}	Maximum or minimum rank of tuple in I_r
y_r, y_s	Sub indexes for W_R or W_S
$h(k_i)$	Single hash function applied on a k_i
L_1,L_2	L_1 holds key value pair for ASketch, L_2 is CMS
N_c,O_c	N_c is the new count and O_c is the old count

hash-based field grouping. This approach help to reduces the load imbalances among worker process, however, creates a huge overhead for tuple accumulation from multiple processing instances for stateful computation. Zhang et al. [22] discuss the stream join in DSPS and state that hash-based data routing strategies create load imbalance and shuffling generates redundancy in join computation. They propose a exponential counting mechanism that keeps record for heavy hitters in the stream and use shuffling for frequent items and and hashing for non-frequent keys in the stream. In our proposed study we use POTC that partitioned the high frequent for separate indexing to balance the load among workers and use random choice for all other keys [45].

3. Preliminaries

In this section, we explain stream join processing, augmented filter, and data partitioning in distributed stream join processing.

3.1. Stream join processing

We provide the definitions of terms that have been used for the join of continues data stream.

Definition 3.1. *(Streaming data): is a continues flow of infinite data items known as tuples $T = \{t_1, t_2, t_3 \ldots\}$. Each tuple t_i consist of finite set of keys and values pair $V = \{\langle k_1, v_1\rangle, \langle k_2, v_2\rangle, \langle k_3, v_3\rangle, \ldots, \langle k_n, v_n\rangle\}$, where $t_1 = \{(k_i, v_i), i \in \{1, m\} \wedge v \in f(k_i)\}$, where $f(k_i)$ represent some value associated with key. However, for stream semantics, the tuple and key are used interchangeably.*

Definition 3.2. *(Sliding window): For a data stream a sliding window W can be defined as the ordered set of recent tuples bounded by time or count. Let considers a time-based sliding window $W_d = \{t_d, t_{d-1}, t_{d-2}, \ldots, t_{d-n}\}$ where a new tuple t_i is added into the beginning of sliding window W_{d+1} and older tuples (expired) T_r are removed depend on the threshold ∂ as explained, $T_r = \{(\{W_d\} \backslash t_{di}^{W}), i \in \{1, n\} \wedge d > \partial\}$. If d and δ are timestamps (resp. counters), we have a time-based sliding window (resp. a count-based sliding window).*

Definition 3.3. *(Stream join) is a tuple of four items $\langle W_R, W_S, \bowtie_\theta, t\rangle$.*
W_R : streaming window for data stream R.
W_S : streaming window for data stream S.
$\bowtie_\theta$: is a join condition between streaming windows $W_R \bowtie_\theta W_S$.
t : a new tuple that can be defined as $t = \{(r, s) \mid r \in R \vee s \in S\}$

A join result $\langle r, s\rangle$ contains all tuples pairs that satisfy θ. Let's considers a case where a new tuple r arrives, the stream join process contains these step.

- Searches W_S to evaluate the theta condition.
- Insert r into W_R.
- Remove expired tuples from W_R or W_S.

The total cost (C) of stream join process for a new tuple can be defined by Eq. (1).

$$C_T = C_{Searching} + C_{Insertion} + C_{Deletion} \tag{1}$$

Indexing stream window items is used to accelerate the stream join, in-memory indexing data structures I_r and I_s are used for streaming windows

W_R and W_S. However, there is an extra overhead of index maintenance along with streaming tuple insertion and deletion. As Shahvarani et al. [3] uses the B$^+$tree for indexing and explains the index maintenance cost as described by Eq. (2), where I_c is the total indexing cost, h_b is the height of B$^+$tree, and λ is the cost associated with the search, delete, or insertion.

$$I_c = h_b.\lambda_{searching} + h_b.\lambda_{deleting} + h_b.\lambda_{insertion} \tag{2}$$

Definition 3.4. *(Index update): a rank R_k is associated with tuple t_i with respect to its location in the indexing data structure I_r. The R_k for each key is updated upon the arrival of new distinct keys. The index update I_u is calculated as the difference between the maximum and minimum R_k for a key during stream processing in particular window W_R as explained by Eq. (3).*

$$I_u = t_{R_k}^{max} - t_{R_k}^{min} \tag{3}$$

In Bi-Stream [17] the indexing data structure is decomposed into sub-indexes $I_r = \{y_{r1}, y_{r2}, \ldots, y_{rn}\}$ and they are updated with respect to time. For low skewed data, the sub-indexes have more distinct items, however, with an increase in Zipf same keys occur more frequently and repeat on many sub-indexes such as $t_i \in \{y_{rj} \mid y_{rj} \in y_r \wedge 1 \geq j \leq n\}$, where t_i is a frequent tuple and I_{rj} shows the total sub-indexes where t_i can exist. We call this tuple t_i as a *redundant key.*

3.2. Augmented sketch

Augmented sketch (ASketch) [23] is built by combining the basic count-based filter and count-min sketch (CMS). It is used both for frequency estimation and finding the top-k frequent items of streaming data. The filter $L_1 = \{k_1, k_2, k_3, \ldots, k_n\}$ is used to keep track of finite top-k elements, similarly, CMS, L_2 estimate the frequency of keys with the help of hash functions $H = \{h_1, h_2, \ldots, h_n\}$ where h_i represents a single hash function. [29]. To keep track of the dynamic popularity of a key k_i in a stream, ASketch uses the swap operation S_o among L_1 and L_2. The L_1 is composed of a new count counter N_c and an old count counter O_c. The N_c keeps the estimation (over-estimation depends on swaps) of the k_i and the difference between N_c and O_c depicts the actual frequency of k_i. The procedure of ASketch is explained by several steps;

- A K_i can only be inserted into L_1, where L_1 has enough space ϕ to accommodate the new request and formally it can be stated as $k = \{k_i \mid (k_i \in K), |L_1| \leq \phi\}$.
- For a case $k = \{k_i \mid (k_i \in K), |L_1| > \phi\}$ the hash functions $H(k_i)$ are applied on k_i to find its position in CMS L_2.
- The layer L_1 uses the priority queue for streaming items. Therefore, a key k_i with the least N_c must always exist on the top of the queue. The swap operation S_o is initiated when the frequency estimation of k_i on L_2 exceeds the N_c of k_j that exists on top of the priority queue. It can be written as $S_o = \{\exists k_i \mid (k_i \in K), minH(k_i) > L_1^{k_{top}}\}$.

3.3. Data partitioning

A DSPS follows the *controller-worker* architecture. The controller controls the data processing, distributed nodes, and job allocation among cluster nodes, similarly, client servers are computing nodes $N = \{n_1, n_2, n_3 \ldots, n_n\}$ that process the continuous data in parallel. The computing nodes have worker processes that are composed of a set of **processing elements** or tasks running distributive to process the data in a real-time $PE = \{pe_1, pe_2, \ldots, pe_n\}$. Toshniwal et al. explain the semantics of data processing of a DSPS in [42] Section 2. A DSP application is represented by the DAG $G = (V, E)$, where nodes V represent the data processing and edges E show the data flow among nodes. Data flow among two processing nodes (source node S_v, destination node D_v) requires pattern (e.g., shuffle grouping [42], partial key grouping [32], key grouping [42], and etc.) known as data partitioning.

An efficient data partitioning strategy depends on the business logic at destination processing element pe_i. Suppose a pe_i is splitting the words from the text sentences then the round-robin strategy is better suited and create no load imbalance. However, for data aggregation such as counting the terms for a particular period of time, the shuffle partitioning strategy creates a large overhead of tuple accumulation from several processing elements for the completeness of the result. Hashing the particular key and selecting the processing task bases on the hash function reduces this accumulation cost, however, creates some overhead of load imbalance among processing elements. Similarly, for distributed join computation, the processing elements keep the windows of streaming data and perform join computation. Appropriate data partitioning for join computation that creates less accumulation overhead and low load imbalance among processing tasks is challenging.

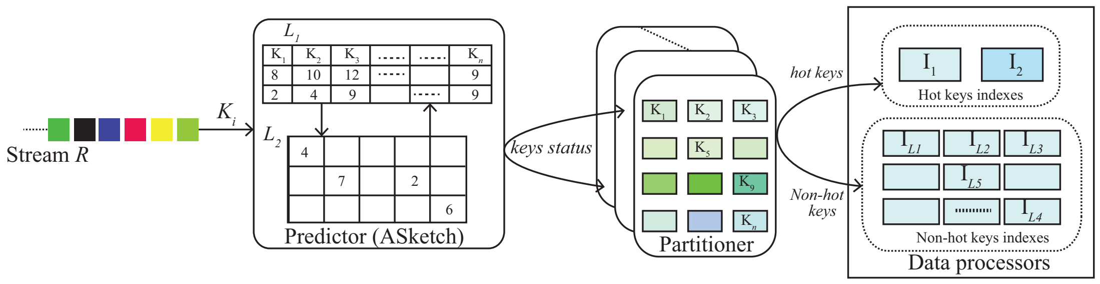

Figure 7: Architecture for stream-aware indexing

4. Stream-aware Join Processing

In this section, we provide details about architecture for stream-aware join processing that includes: use for stream predictor for data partitioning, data distribution among processing nodes, use of separate indexing for hot and cold items of streaming data, and dynamic change in popularity of key during stream processing. The architecture of the proposed stream-aware index solution is shown in Figure 7.

4.1. Stream predictor

We employ ASketch [23] to keep track of data popularity. As discussed in Section 3 the ASkecth consists of two layers. The first layer L_1 keeps the record for hot items, whereas the second layer L_2 keeps non-hot keys. In the stream join application, a special vertex of DAG is used to build ASketch. This vertex lies adjacent to the source streaming node. Every key of the streaming data enters into the ASketch as depicted by the predictor of Figure 7. All keys that belong to layer L_1 are considered hotkeys and treated differently by the data partitioner than keys that belong to the CMS L_2. Algorithm 1 explains how we take advantage of ASketch for keeping track of hot and cold keys in streaming data.

The input to Algorithm 1 is the streaming data, fixed size of the filter, and CMS size. Initially, a key k_i enters into L_1 of ASkecth and then linearly scans the filter to check the presence of k_i in L_1 as depicted by line 1 of Algorithm 1. Similarly, from line 2 to line 4 show that the item k_i exists in L_1 where relevant N_c is incremented by 1, and return k_i as a hotkey, otherwise, insert the key k_i into the filter L_1 with $N_c = 1$ and $O_c = 0$ and return non-hot key as described by line 5 to line 7 of Algorithm 1 . However, in a case where the filter has not enough space to accommodate the new k_i then the key k_i

Algorithm 1: Tuples prediction with ASketch

```
   Input: Key ($k_i$), $L_1$ size ($\phi$), and $L_2$ (HxW)
   Output: $k_i$ popularity
1  Scan the layer $L_1$ linearly;
2  if $k_i \in L_1$ then
3      update $N_c$ to $N_c + 1$;
4      return hot($k_i$);
5  else if $| k_i | \leq \phi$ then
6      insert $k_i$ in $L_1$;
7      return non-hot($k_i$);
8  else
9      insert $k_i$ into CMS;
10     if min Hf($k_i$)> $L_1^{k_{Top}}$ then
11         update $L_1^{k_{Top}}$ with min Hf($k_i$);
12         insert $L_1^{k_{Top}}$ frequency $N_c - O_c$ in $L_2$;
13         return hot($k_i$);
14     else
15         return non-hot($k_i$);
```

is inserted into the adjacent L_2 of ASketch. Layer L_2 has a CMS, with hash functions and width of the sketch. The k_i is inserted into the hash locations of CMS and returns a non-hot key for data partitioning as shown by line 15 of Algorithm 1. The swap operation is initiated once for those keys whose estimated frequency by CMS exceeds the smallest N_c of the $L_1^{k_{Top}}$ in L_1. The filter is updated by k_i and the difference between the new count and the old count is added to the hashed location of the CMS. This key is considered as a newly added frequent key and treated as a hotkey by the data partitioner as depicted by line 9 to line 13 of Algorithm 1.

4.2. Stream partitioner

We present a novel data partitioning strategy where the result from the predictor acts as an input for the data partitioner. The partitioner knows all the active processing tasks for the application and exploits the key-count esteems provided by the predictor for routing the tuples to computational

Algorithm 2: Data partitioning for join processing

```
   Input: PEs, Algorithm1 output $\langle k_i, K_i status \rangle$
   Output: $k_i \mapsto pe_i$; $i \in \{1, n\}$
1  if $k_i status == Hot(k_i)$ then
2      right $(c_i) \leftarrow 0$;
3      current $(c_j) \leftarrow 0$;
4      choices $C = \{pe_{n-1}, pe_{n-2}\}$ ;                // PEs can be added in $C$
5      while $True$ do
6          $c_i \leftarrow c_j + 1$;
7          if $c_i < |C|$ then
8              return $C[c_i]$;
9          else if $c_i == |C|$ then
10             $c_j \leftarrow 0$;
11             return $C[pe_{n-1}]$;
12 else
13     available choices $C = \{pe_1, pe_2, pe_3, \ldots\ pe_{n-3}\}$;
14     return $k_i \leftarrow$ rand$\{C\}$
```

nodes, as shown in Figure 7. These PEs keep incoming keys as a stream window and build the indexes for those keys for future join processing. The data partitioner queues the incoming data stream items and route the hotkeys to two processing elements uniformly. As Nasir et al. [32] explain that using two choices instead of one improves exponential improvement for load balance than single choice. However, increasing these choices do not have too much impact on load balancing. Moreover, increasing choices increases the aggregation overhead as discussed in Section 7. These two PEs are completely isolated for non-hot keys of the stream and they hold the subset of the streaming window (hot items). So, all of the non-hot key in the stream are routed with a randomized approach that follows the well-known *super market* model [45, 46]. In details, we assume that the tuples arrive with Poisson stream rate λ_n for PEs where $\lambda < 1$; then, each non-hot key k_i chooses m processing tasks uniformly at random from $n - 2$ processing tasks [45]. We empirically analyze the impact of random choice with varying skewed data and compare the performance against hash-based key routing and round-robin strategy in Section 7.

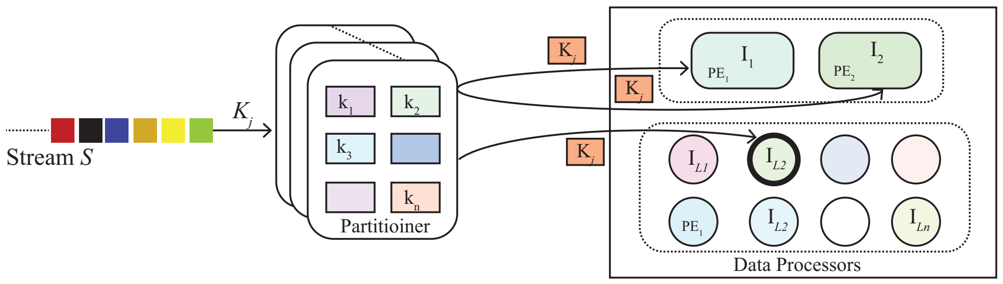

Figure 8: Data partitioner for stream join

Algorithm 2 provides the complete procedure for data partitioning for stream data windowing. The input to the algorithm is available processing tasks and the output of Algorithm 1 $\langle k_i, K_i status \rangle$, where $K_i status$ indicate the key popularity in the stream. The output of the Algorithm 2 gives the mapping for the key to the processing tasks $k_i \mapsto pe_i$. Our approach involves two PEs for hotkeys, so the size of the choice set is two as depicted by line 4 of Algorithm 2. However, more processing tasks can be added to the choices C. Moreover, a while loop is used for assigning the key k_i to the processing tasks. For every new key k_i, the choice will be incremented by one, however, at the end of the choices the new key will be allocated to the first. This whole assignment procedure is described by lines 5 to line 11 of Algorithm 2. Similarly, for all non-hot key items, a key k_i is mapped randomly from the set of available PEs as depicted by line 13 and line 14 of Algorithm 2. Each PE maintains a sub-window of the complete sliding window.

Similarly, the data partitioning strategy behaves differently against streaming tuples for join. Instead of routing the lookup key k_j to a single processing element, this partitioner broadcast the k_j to the two processing tasks that hold the sub-window of hotkeys as shown in Figure 8. Moreover, the key k_j has also been evaluated for non-hot keys. A random choice has been applied to find the respective processing element that holds the state of the sub-window. The streaming window is divided into multiple sub-windows and share their respective state among the poll of the non-hot key processing element. A detailed description about the shared windowing structure is discussed later in this section.

4.3. Stream window Indexing

Queries that involve non-equality predicates require an efficient indexing structure to accelerate the query processing, however, it creates the overhead

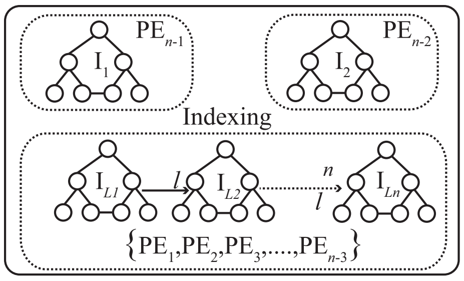

Figure 9: Indexing window contents among PEs

of index update. The overall performance improvement with indexing eliminates this cost [3]. In DSPS, the keys are distributed among PEs, where the contents of streaming data exist.

We use a B$^+$tree for indexing the window contents. Separate indexing data structures (I$_1$ and I$_2$) are considered for hotkeys that are maintained by two PEs of DSPS as depicted by the data processors of Figure 7. Similarly, all cold items use a single linked chain of B$^+$tree (I$_{Li}$), that dynamically share its state among many processing tasks. The indexing window contents among processing elements can be depicted in Figure 9. However, using separate indexing data structures for every task can create overhead of tuple accumulation for the completeness of results. Each k_j from stream S needs to check every processing task, whereas these PEs exist on many nodes of the cluster that create a huge overhead over the network. Our proposed solution utilizes the concept of *check pointing* for non-hot keys where PEs share their state of data structure after regular intervals of time. This process also creates an overhead of checkpointing and can return an incomplete result, however, we measure the system performance against synthetic increasing skewed data in Section 7, where we observed that the proposed approach is performing better with increasing Zipf for completeness of stream join results.

Each sub-index of I$_{Li}$ is linked with other successive sub-index using a linked list as shown in Figure 9. In the stream join, the update operation is very frequent along lookup, so sub-indexes reduce the cost of index update than using a single index structure [17, 3]. A new tuple can only be added into the I$^a_{Li}$, whereas the whole I$^r_{Li}$ is removed from the shared linked list. Moreover, the key popularity can be changed during stream processing, yet it can be handled as follows:

- If non-hot key at any instance becomes popular during flow of stream, then a newly arrives key is indexed into the PEs dedicated to hotkeys. This procedure creates the redundancy of such keys in several indexes of streaming windows. However, we empirically observe that using separate indexing data structures for hotkeys reduce the rate of redundancy on sub-indexes for increasing Zipf as depicted by Figure 6.

- If hot key frequency is changed to un-popular during stream processing then key is removed from the indexing structures of hot key elements and inserted into the shared-linked B$^+$tree. This procedure introduces a little overhead of tuple removal and adding into the sub-index data structure, however, as Roy et al. [23] explains that increasing Zipf has a little tuple swapping overhead.

4.4. Stream join

The stream join requires two steaming windows W_R and W_S of stream R and S as depicted by Figure 1. A new tuple t_i from any stream needs to evaluate the predicate ($\bowtie_\theta$) against the opponent streaming window. Stream inequality join predicates ($\leq, \geq, <, >$) usually have a high selectivity rate, so indexing the streaming windowing item speed up the lookup procedure. Our proposed indexing structure is shown in Figure 9. Moreover, stream join in stream processing requires a regular update to the streaming window and its index on new tuple arrival t_r and t_s. Similarly, it also includes the deletion of expired tuples based on time or counts from the streaming windows.

We create a two-way indexing structure (i.e, hotkey indices and non-hot key indices). Let's consider a case where a tuple k_r from stream R arrives to evaluate the predicate θ. It performs lookup operations in W_S indexing data structures. The contents of the sliding winding W_S exist on many PEs, so k_r explores the dedicated hotkey indexed processing tasks and add the results into the result set. Moreover, t_r also exploits the linked-indexing structure because a subset of tuples from W_S also exists in these PEs. The t_r explores the shared linked list and their respective sub-indexes and adds the result to the global result set. The t_r exploited the processing tasks for both hot keys and non-hot keys in a parallel fashion on distributed nodes.

In stream window join new tuple need be to be added on its respective window for future join. We keep track of the hotkeys and separate them from non-popular keys. In the case of the hot tuple, t_r is indexed in PEs that are dedicated to hotkeys in a round-robin fashion. The indexing data structure on these PEs is updated on the arrival of the new key. Similarly, all non hot are indexed to the active sub-index of the linked tree. The update procedure is applied to the tree of active sub-index. This procedure reduces the cost of index updates on sub trees structure on processing tasks.

Tuples deletion from the streaming window is an essential part of stream join. Individual tuple deletion from the indexing tree of the streaming window increases the overhead of index updates. Tuples deletion from the streaming window depend on time or count. Our proposed model uses a separate index data structure for the streaming window so deleting the expired tuples from each separate indexing structure is also expensive. We propose a novel deletion method for tuple removal. The hotkeys are removed from the dedicated indexing trees during their change in popularity and inserted into the active tree of the shared linked indexing structure. Similarly, the coarse-grained tuple deletion methodology is adopted for deleted of tuples from the sliding window. As analogous to BiStream [17] and PIM-tree [3] the whole sub-tree is removed from the indexing structure on expiration.

For a complete evaluation of the predicate, the tuple from the other stream t_s is also explored against streaming window W_R. For example, a predicate condition is $(t_r \leq W_S)$ where tuples for the streaming window W_R are less or equal to the tuples in the streaming window of W_S. A new tuple where $t_s > W_R$ also fulfills the existing predicate for stream window join. All tuples of W_R need to be evaluated against t_s. Stream join algorithm processes this predicate and added the result into the result set against the query on the continuous data. An analogous stream-aware join procedure is applied for this tuple join processing.

The cost of the stream join includes the lookup, update, and delete of the cost of indexing data structure. Moreover, we also use ASketch which also costs during tuple popularity prediction. The lookup cost includes the traversing cost of all indexing tree structures that hold the streaming window contents such as hotkey indexers I_1 and I_2 and shared linked index I_{Li}. The cost of lookup is depicted by Eq. (4).

$$C_{lookup} = I_1 + I_2 + \sum_{i=1}^{n} I_{Li} \tag{4}$$

Similarly, a new t_i can only be inserted into the index structure of hotkeys or active sub-index of the linked tree. This cost is represented by the Eq.(5). If a new key is hot then only one sub-index on pe_i that is selected by the partitioning strategy is updated, moreover, in the case of a non-hot key, the linked active sub-index I^a_{Li} is updated from the range of linked trees.

$$C_{update} = \begin{cases} 1/2, & \text{if } k_i \text{ is hot key} \\ 1/n, & \text{otherwise, n is total sub-trees in } I_{Li}. \end{cases} \quad (5)$$

Expired tuples from the sliding window are removed during the insertion of new key t_i. The cost of removing a tuple from hotkey indexing is the same as for updating, however, this key t_i is also added to the linked data structure that has updated cost in its active sub-index. The total cost of k_i removal from hotkey is $1/2 + 1/n$. Moreover, we opt for the coarse-grained tuple deletion methodology for the whole sub-index I_{Li} [17, 3].

ASketch also increases the overhead during tuple prediction. However, the performance gain due to this sketch is much higher than the cost of its prediction. We perform a detailed experimental evaluation on synthetic and real-world data to measure the performance of by using ASktech and show the superior performance of stream join algorithm on high selective queries in Section 7.

5. Stream Join Solutions for DSPS

In this section, we provide the implementation details for the state-of-the-art solutions of stream join algorithms in DSPS. The DSPS consists of distributed operators that process the input stream to a number of processing tasks that operate in a parallel fashion. In this section, we map the semantics of exiting stream-join solutions by considering the DSPS operators as explained by Algorithm 3.

5.1. Broadcast join

Broadcast join is widely used in Spark batch processing join [10] where the smaller side of the data set is broadcast to all executors of the cluster nodes and performs the join from the candidates of other tables. This processing semantic is much faster than shuffle partition due to its standalone execution on a single partition for whole predicate evaluation. However, for large data set this scheme create a serious memory overhead, moreover, for non-equi join where it increases the latency while completing the θ evaluation.

In this work, we implement the broadcast join in Storm [9], where we use *all grouping* for broadcasting the k_i to all processing tasks of the DSPS application as depicted by the line 3 of Algorithm 3. This process will create multiple copies of the W on several nodes. To search a k_j, it is hashed using the hash function to find a pe from a set of tasks and then perform the query evaluation using for loop as described by line 4 of Algorithm 3. (We named this *Broadcast join* to *Broadcast hash join* (BCHJ)). Similarly, a tuple is removed from all streaming windows in each pe depending on the tuple removal threshold as explained formally by line 5 of Algorithm 3.

Let considers the cost of inserting a k_i on a single processing element as c_i^{pe} then injecting a k_i to n processing tasks can be represented by $n(c_i^{pe})$. Similarly, to delete an key k_i from an array of PEs, broadcast hash join has the same cost as of insertion. However, search requires the cost of hash computation and exploring a window W_i^{PE} of particular processing tasks from start to end for a non-equality predicate. Thus total cost can be represented by Eq. (6).

$$C_{BCJ}^{k_i} = 2n(c_i^{pe}) + H(k_i) + \sum_{i=1}^{n} W_i^{PE} \tag{6}$$

5.2. Split join

Split join [19] divides cores of a processor for dedicated streaming windows W_R and W_S. A stream item k_i needs to expedite all cores. Following the semantics of the multi-core system, we use processing tasks instead of cores. Each pe hold both streaming items r and s for completeness of the join predicate. Individual processing tasks hold the subset of streaming items for each window. Tuple accumulation from all PEs creates an overhead for a large number of tasks, however, smaller processing tasks increase the update cost of index-tree structure for non-equality join.

In this work, we implement the *Split join* using the configuration of Storm. We use *shuffle grouping* for k_i that routes the tuples from source to the destination processing tasks in a round-robin way as depicted by line 7 of Algorithm 3 and insert the key into its local index structure. Similarly, k_i uses the *all-grouping* for tuple partitioning to evaluate the predicate against all local windows of processing tasks as described by line 8 of Algorithm 3. Moreover, line 9 shows the single tuple t_i removal from the tree structure during insertion depends on a certain threshold. The cost of the insertion includes finding the pe and insertion as discussed above. We use the same cost of tuple deletion as the insertion of key t_i. Both insertion and deletion will update the indexing structure. Moreover, the search includes all cost of exploring the local windows of PEs, so the total cost for a split join can be

Algorithm 3: Stream join algorithms for DSPS

```
Input: $\{PEs\}$, $k_i$, $W$, $\theta$ condition
Output: $k \bowtie_\theta W$
1  switch Strategy do
2      case BCHJ do
3          Insertion: $k_i \mapsto \forall\{PEs\}$ ;                // same $W$ on each PE
4          Look up: $PE_1 \leftarrow H(k_i)$ ;              // H to select $PE$ for $\theta$ evaluation
5          Remove: $\forall\{PE_i^W\}\backslash k_i$ ;   // Remove $k_i$ depends on threshold
6      case Split-join do
7          Insertion: $k_i \mapsto \{pe \in \{PE\}|$ pe is selected round robinly$\}$;
8          Look up: $k_{i\theta} \rightarrow \forall\{PEs\}$;
9          Remove: $(k_i \in \mathbf{pe})\backslash k_i$
10     case Chained-Index do
           // shared a linked sub index among all $PEs$
11         Insertion: $k_i \mapsto \{y_s \in \{I_c\}|$ y is sub-index of $I_c$ $\}$;
12         Look up: $k_{i\theta} \rightarrow \forall y_i^n : i \in \{1$ to n$\}$;
13         Remove: $I_c\backslash y_s$: $y_s > \lambda$ ;                    // $\lambda$ is threshold
14     case PIM-Tree do
           // shared a linked sub index among all $PEs$
15         Insertion: $k_i \mapsto \{y_s^r \in \{I_c^r\}|$ $y^r$ is ranged sub-index of $I_c^r\}$;
           Look up: $k_{i\theta} \rightarrow \forall y_i^r : i \in \{1$ to n$\} \cup I_{merged}$;
16         Remove: // coarse-grained tuple removal
17         if $\lim |I_c^r| > \omega$ then
               // $\omega$ is threshold for merging
18             Merge $\forall y_s^r$ to $I_{merged}$;
19             $\forall k_{i\rightarrow n} > \gamma \ \backslash\{I_c^r, I_{merged}\}$; // $\gamma$ is threshold for removal
20
21     otherwise do
22         exit
```

described by Eq. (7).

$$C_{Split-join}^{k_i} = (x + 2(c_i^{pe})) + \sum_{i=1}^{n} PE_i^w \tag{7}$$

5.3. Chained-index

Chain-index [17] is implemented for Apache Storm, however, its implementation details are missing from the description [17]. The contents of this sliding window are indexed into the number of sub-linked chain indexes. These indexes are distributed among many PEs, moreover, the threshold for the whole index update is based on the time. Many data items of the stream are very often repeated on many sub-indexes, moreover, small-size sub-indexes for larger windows increase the overhead of index updates.

In this work, we use *field grouping* for tuple partitioning to processing tasks. A stateful linked list of indexing trees is shared among PEs that is updated after regular checkpointing. The completeness of the results depends on the checkpoint interval. A tuple is inserted into a local list of PEs, whereas after several intervals of time, the state of all linked lists is shared to a single list. Similarly, a coarse-grained tuples are removed, where the whole sub-index is removed from the linked list.

The insertion cost of the chain-indexed for DSPS includes the cost of hash computation for the selection of processing element and the sub-index updating cost to c_y. Moreover, we use coarse-grained tuple removal that can be considered negligible [3]. The cost of searching for a non-equi join predicate includes the search of whole sub-indexes. The checkpointing factor also affects the total cost of the chained index. The cost can be represented by Eq. (8) where ρ is the cost for checkpoints that depend on time.

$$C^{k_i}_{Chained-index} = (H(k_i) + c_y) + \sum_{i=1}^{n} y_i^r + \rho_{time} \tag{8}$$

5.4. PIM-Tree

The partitioned in-memory merge tree is designed for multi-core systems. It is a two-layered data structure I_p^1 and I_p^2 that includes mutable and immutable components holding the contents of sliding windows. Initially, the stream items are indexed using the same concept of the chained index, however, after the merge operation, the ranges of sub-indexes are redefined based on the depth of the single immutable index tree.

In DSPS, we use a similar methodology of the chained index for mutable component, by using a single hash function for tuple routing among processing tasks and use a shared linked list, however, a dedicated data structure on single *pe* hold the immutable indexing structure of PIM-Tree. The mutable index tree is merged into the immutable tree where tuples are deleted during

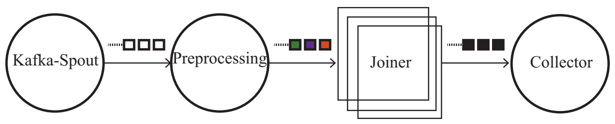

Figure 10: Topology view for join processing

the merge operation. The contents of the shared linked list among processing tasks are merged into the immutable data structure. A newly inserted tuples wait during the merge operation.

The cost of new tuple insertion is the same as the cost of the chained index, moreover, it also costs checkpoint intervals where linked sub-trees share their state. Similarly, a search operation for a non-equality predicate is required to explore both mutable and immutable components. The tuples are deleted in a batch during the merge operation so the cost of tuple deletion and the merging cost M of mutable to the immutable tree is the total deletion cost, checkpoint overhead is also included where the tuples in different PEs share their state of linked list. The total cost can be explained by Eq. (9).

$$C^{k_i}_{PIM-tree} = (H(k_i) + c_y) + (\sum_{i=1}^{n}(y^r_i) + c_I^{Immutable}) + M + \rho_{time} \tag{9}$$

6. Experimental Setup

In this section, we provide a discussion about cluster setup, data sets, and metrics used for the evaluation of stream inequality solutions in DSPS.

6.1. Cluster setup and topology

The inequality join algorithm is implemented over the top of Apache Storm DSPS. The cluster consists of 10 distributed machines with heterogeneous computing resources. The machines run Kafka for input data, Zookeeper for coordination among computing nodes, Nimbus for Storm master, and various Supervisors as worker processes. The streaming data is consumed by the storm cluster using Kafka spout and forwarded to the other bolts for join processing. The illustration of the storm application is depicted in Figure. 10.

6.2. Data sets and predicate description

We employ two real data sets and one synthesize data set for the evaluation of the inequality join algorithms.

- The NYC taxi data set, from the DEBS grand challenge 2015 [27]. This data set contains information regarding taxi trips with 17 various fields and 173 million events. We consider the $fare$ as metric for indexing and evaluate a single predicate with a self-join as depicted by Query 2.

  Query 2: Query for NYC Taxi data.

  ```
  SELECT *
  FROM NYCTaxi T1 JOIN NYCTaxi T2
      ON T1.FARE > T2.FARE
  WINDOW w AS (RANGE INTERVAL ‘W_d’ SECOND PRECEDING);
  ```

- The BLOND data set contains the continuous measurement of voltage V and current I of aggregated circuit inside an office measurement [26]. Run time analysis can help with efficient load profiling, power saving, and automation. We use BLOND 250 data set where each file contains three pairs of V and I readings of three individual circuits. The size of the data set is 23 TB and we use some chunks of files for the evaluation of the inequality join algorithm. In this case, we evaluate the inequality join algorithms against two predicates in the same topology as depicted by Query 3:

  Query 3: Query from Example 1.1

  ```
  SELECT *
  FROM R JOIN S
      ON R.V1 > S.V2 AND R.I1 < S.I2
  WINDOW w AS (RANGE INTERVAL ‘W_d’ SECOND PRECEDING)
  ```

- We also evaluate these inequalities join algorithms against synthetic Zipf data ranging from Z=0.1 to Z=2.0 that contains 32 million tuples. We adopt a single predicate and use self-join for evaluation as shown by Query 4.

  Query 4: Query for Zipf data.

  ```
  SELECT *
  FROM ZIPF Z1 JOIN ZIPF Z2
  ```

```
     ON Z1.T > Z2.T
WINDOW w AS (RANGE INTERVAL ‘W_d’ SECOND PRECEDING);
```

6.3. Performance metrics

We evaluate the inequalities join algorithms and proposed a partitioning strategy against various performance metrics.

- **Load Imbalance**: The difference between the maximum and average load among PEs (i.e., the number of tuples processed by the PE at a given time).

  $$load\ imbalance_{PE} = max(load) - avg(load)$$

  Similarly, **aggregation** is defined as a collection of partial results from various PEs for completeness.
- **Throughput**: The number of tuples processed per second:

  $$throughput = \frac{\#\{processed\ tuples\}}{time}$$

  In this case, we measure the throughput for insertion and search operation for each new tuple into the sliding window.
- **Latency**: The time taken by the tuple for complete predicate evaluation. We define the latency as the time difference for tuple t_i from pre-processing node to collector node of DAG of DSPS as depicted by Figure 10.

  $$latency_{t_i} = time_{t_i}^{collector} - time_{t_i}^{preprocessing}$$

- **Completeness**: When an inequality join is performed over a windowed stream of data, the result might differ from the actual result that we would get in a batch setting [47]. This is due to time latencies introduced by the network or straggling computational nodes. To measure that, we define the *completeness* as:

  $$completeness = 1 - \frac{|X - \tilde{X}|}{|X|}$$

where $\tilde{X}$ is the output of the stream join and X (expected result) is the output generated by performing the same inequality join in the ideal setting with no latencies. *Completeness* varies in the range $[0, 1]$ and represents the fraction of joined tuples over the existing ones, a value close to 1 is better.

We evaluate the inequality join algorithms with varying single parameter and fixed others such as input tuple rate, number of tasks, and the sizes of the sliding window W_d.(The fixed size of parameters are input rate=2ms, processing elements=40, window length =50sec). We evaluate our method only for time-based sliding windows, however, our solution can easily be extended for count-based windows. Our comparison takes into consideration relevant partitioning schemes and other join algorithms for each evaluation metric. For example, load imbalance is only a concerned with certain types of partitioning schemes, such as hash-based data partitioning, while the round-robin strategy distributes data uniformly among downstream instances. Similarly, the random choice of partitioning scheme can have an impact on data accumulation costs for stateful data operations, while the hash-based strategy incurs negligible data accumulation costs.

Our stream join algorithm has one overhead of tuple prediction. We employ ASketch [23] which has two layers for keeping track of hot-key elements. Roy et al. [23] state that throughput for the tuple processing algorithm is gradually increasing with increasing Zipf values in Section 7 [23]. Higher Zipf values have less swapping of keys among layers of the augmented sketch. We adopt the same setting as top-32 items for L1 and 16KB of CMS for tuple prediction.

7. Results and Discussion

In this section, we provide a discussion of the experimental results for the proposed stream-aware solution and other state-of-the-art stream inequality join solutions. For ease we abbreviate different inequalities solution as follows: broadcast hash joins as (BCHJ), split Join (SJ), multi-chained index (MCI), Chained index (CI), Partitioned in-memory merge tree (PIM), and proposed Stream-Aware stream inequality as (STA). Multi-chain index is another variant of chained index, however, in MCI the new key k_i is indexed in all processing instead of single processing element, moreover, it also eliminates the checkpointing for state synchronization.

7.1. Performance measurement for real data

Figure 11 to Figure 13 report the insertion throughput for taxi data set with different combinations of parameters. We consider the average throughput with an equal number of tuples processed for each inequality join algorithm. Figure 11 shows the throughput with a variation of the input data rate that is controlled by the Kafka producer. The results demonstrate that the proposed stream-aware solution has 1.6x, 1.13x, and 2.6x superior performance than split join, chain index, and PIM indexing structures. Similarly, for small changes in input data rate, the performance of the proposed stream-aware is superior to the alternative, however, the throughput rate for PIM is gradually improving for low input data rate.

Figure 12 depicts the result for throughput by setting constant configuration for input rate, window size, and changing the processing elements. Results depict that the proposed stream-aware solution has 1.5x, 1.2x, and 2.4x better through than split join, chain index, and PIM indexing structures. For split join the tuples need to search relevant processing elements before insertion, similarly, the PIM tree has two overheads: 1) merge operation and 2) state synchronization among processing elements. The proposed stream-aware solution and chain index only need state synchronization where separate indexing structures for hot key enhance the performance of the stream-aware solution.

Figure 13 shows the throughput result for changing time-based window size. Results depict that for smaller window sizes the proposed stream-aware solution has 2.5x, 2,0x, and 1.6x better than Split join, chain index, and PIM indexing structure. However, for larger window sizes, the performance improvement for a stream-aware solution is decreased due to more tuples accumulation overhead. The throughput performance of the stream-aware solution is superior from a smaller window size to a larger window length.

Figure 14 to Figure 16 show the latency for lookup operation for taxi join data set. Figure 14 reports the results with changes in input data rate. The result depicts that the proposed stream-aware solution has 7x, 5x, 2.3x, 2.8x, and 3.0x better latency performance than the multi-chain index, PIM indexing, chain index, split join, and BCHJ. Similarly, Figure 15 shows the latency for lookup operation with changing processing elements. Figure 15

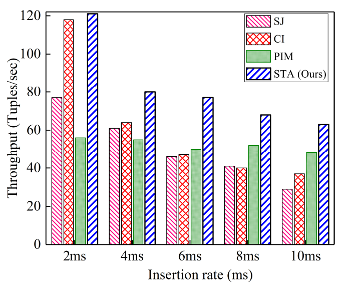

Figure 11: Throughput for taxi data with varying insertion rate

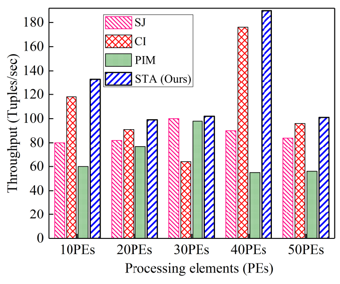

Figure 12: Throughput for taxi data with varying processing elements

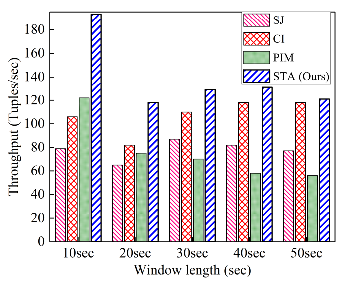

Figure 13: Throughput for taxi data with varying window size rate

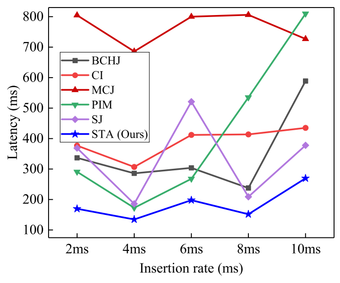

Figure 14: Latency for taxi data with varying insertion rate

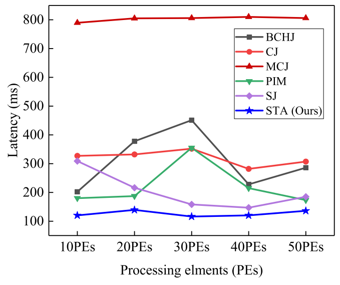

Figure 15: Latency for taxi data with varying processing elements

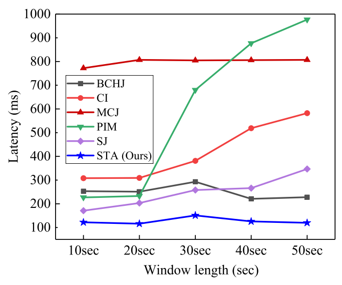

Figure 16: Latency for taxi data with varying window size rate

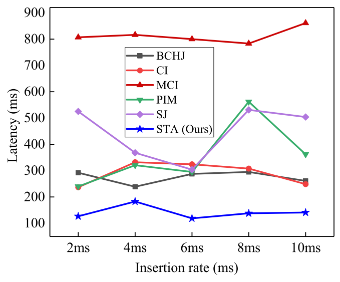

Figure 17: Latency for BLOND data for V with varying insertion rate

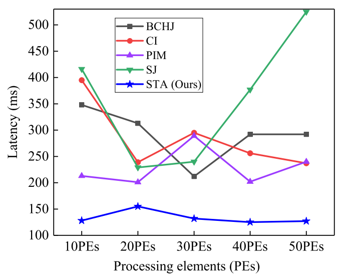

Figure 18: Latency for BLOND data for V with varying processing elements

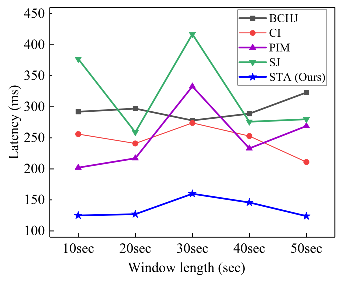

Figure 19: Latency for BLOND data for V with varying window size rate

shows that the proposed stream-aware solution has 3.8x,1.7x,2x,1.2x, and 7x average improved throughput than BCHJ, PIM indexing, chain indexing, split join, and multi-chain indexing solution. Figure 16 shows the latency of the inequality join algorithm with varying window sizes. In this case proposed

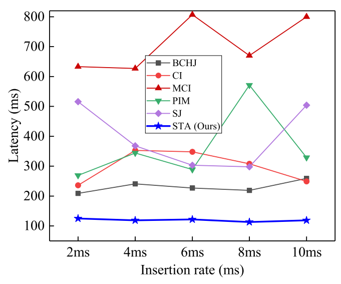

Figure 20: Latency for BLOND data for I with varying insertion rate

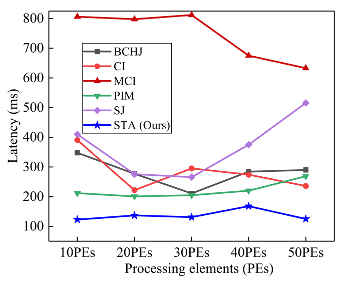

Figure 21: Latency for BLOND data for I with varying processing elements

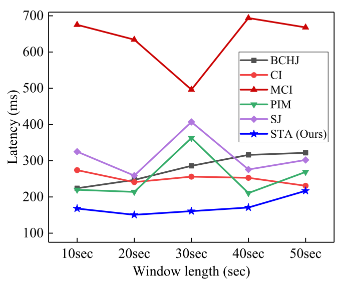

Figure 22: Latency for BLOND data for I with varying window size rate

stream-aware solution has 7x, 5x, 2.5x,3.4x, and 2x better average latency measurement than the multi-chain index, PIM, split join, chain index, and BCHJ.

The query for the electric BLOND data set consists of predicates; 1) for voltage, and 2) for current. Figure 17 to Figure 19 show the latency measurement for electric BLOND data set for voltage predicate with varying parameter setting. Figure 17 shows the latency for lookup operation with varying input data rates. Figure 17 shows the result of changing input rate that depicts that stream-aware solution has 3x, 4x, 4.2x, 4.8x, and 8x time better latency measurement than BCHJ, chain index, PIM index, split join, and multi-chain indexing structure. Moreover, Figure 18 depicts the result with changing processing elements. Our previous observations stated that the multi-chain index has similar latency. For ease, we discard the multi-chain index and observe performance against other state-of-the-art approaches. Results depict that stream-aware solution has an average 2x, 1.5x, 5x, and 2.8x better performance than PIM, chain index, split join, and BCHJ. Similarly, for changing window size, the proposed stream aware has 3.8x, 3x, 2.1, and 2.8x superior performance than split join, BCHJ, chain index, and PIM indexing structure.

Figure 20 to Figure 22 shows the latency for lookup operation for another predicate I of the electric blond data set. Figure 20 shows the lookup performance with varying input rates for the current predicate. Figure 20 shows that the proposed stream-aware solution has 2x, 3.2x, 3.5x, 3.0x, and 7.8x better than BCHJ, chain index, PIM, split join, and multi-chain index solution. Similarly, for changing processing elements the proposed stream-aware solution has 2x, 3x, 3.4x, 4.3x, and 7.3x superior performance against PIM

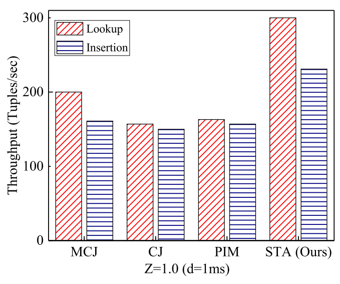

Figure 23: Throughput for insertion and lookup for Z=1.0 and insertion delay d=1ms

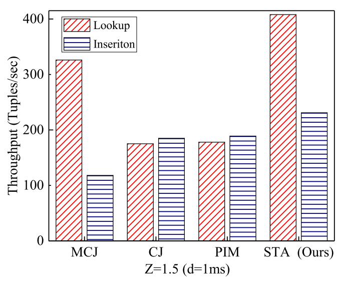

Figure 24: Throughput for insertion and lookup for Z=1.5 and insertion delay d=1ms

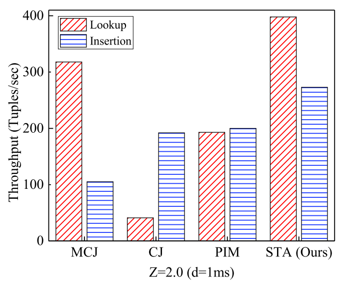

Figure 25: Throughput for insertion and lookup for Z=2.0 and insertion delay d=1ms

indexing, chain index, BCHJ, split join, and multi-chain index join. Similarly, for processing elements the proposed approach is 6.5x,3.1x,2.8x, 2.5x, and 2.2x better than multi-chain, split join, PIM, BCHJ, and CI as depicted by Figure 21. Figure 22 shows the latency performance for the current predicate. The result depicts that the stream-aware solution has 6.5x, 3.1x, 2.8x, 2.5x, and 2.2x better than the multi-chain index, split join, PIM indexing, BCHJ, and chain index solutions.

7.2. Performance measurement for synthesize Zipf data

Figure 23 to Figure 25 depicts the throughput results for skewed distribution that ranges from Z=1.0 to Z=2.0 with insertion delay of 1ms. Each result shows the average insertion and lookup tuple processing throughput for various in-equality join algorithms. Figure 23 depicts that the proposed stream-aware solution has 1.5x, 2x, and 2.1x superior performance than the multi-chained, chained index, and PIM index for z=1.0. Similarly, Figure 24 shows the throughput for Z=1.5 with a delay of d=1ms insertion rate. Results depict that 1.3x, 2.2x, and 2.1x are superior to multi-chained join, chained index, and split join. Figure 23 depicts that stream-aware solution has superior performance than alternatives. Figure 26 to Figure 28 shows the throughput with delay d=2ms. Results depict that the proposed stream-aware solution has better performance the than alternatives. Moreover, Figure 29 to Figure 28 show that proposed approach stream-aware approach has superior performance than alternatives with increasing Zipf values.

The stream-aware solution is performing better due to several reasons. Our solution consists of both data partitioning and stream indexing struc-

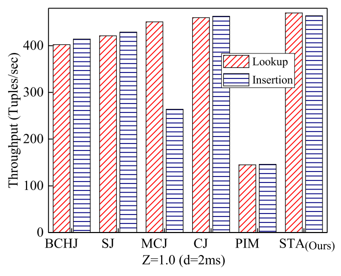

Figure 26: Throughput for insertion and lookup for Z=1.0 and insertion delay d=2ms

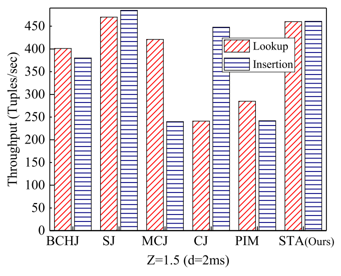

Figure 27: Throughput for insertion and lookup for Z=1.5 and insertion delay d=2ms

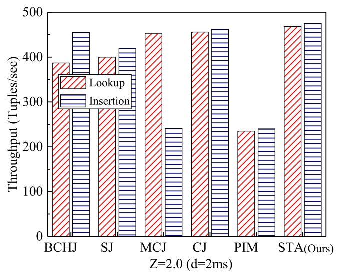

Figure 28: Throughput for insertion and lookup for Z=2.0 and insertion delay d=2ms

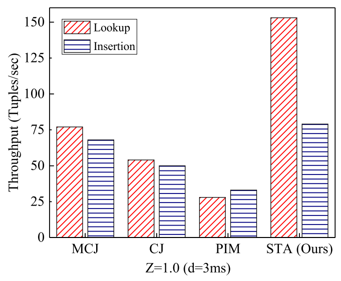

Figure 29: Throughput for insertion and lookup for Z=1.0 and insertion delay d=3ms

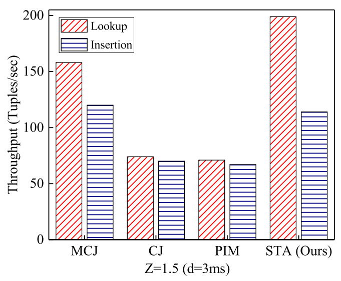

Figure 30: Throughput for insertion and lookup for Z=1.5 and insertion delay d=3ms

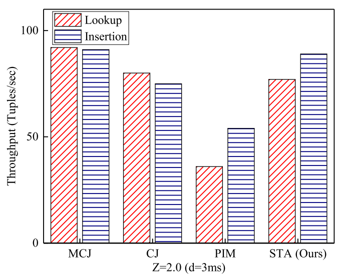

Figure 31: Throughput for insertion and lookup for Z=2.0 and insertion delay d=3ms

ture for a complete stream join process. BCHJ replicates the window to all processing elements which is less memory efficient. Split join requires searching all processing elements in a round-robin way for completeness of results. The chained index needs a lot of effort in state synchronization. Similarly, PIM has a two-way search structure that includes a lot of overhead of merging data structure. Our approach uses a power-of-the-two-choices for hotkeys and requires less state synchronization for the non-hot key. This approach helps for better performance than other approaches.

7.3. Data partitioning and result completeness

The stream-aware solution uses a novel partitioning scheme by augmenting the power of two choices and random partitioning strategies. Figure 32 shows the comparison of load imbalance among PEs for hash-based and random partitioning strategies. The result depicts that with an increase in Zipf

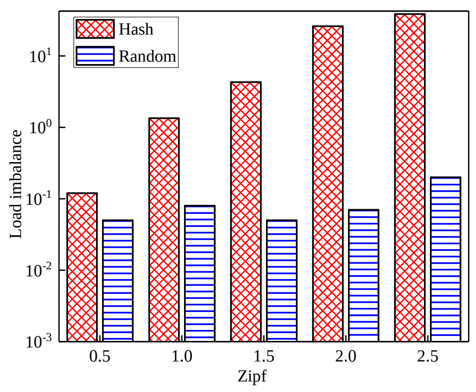

Figure 32: Load imbalance comparison for hash and random data partitioning (the lower the better).

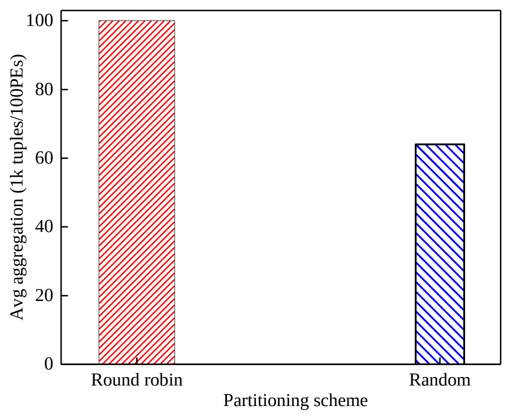

Figure 33: Aggregation of tuples for round-robin and random partitioning schemes (the lower the better).

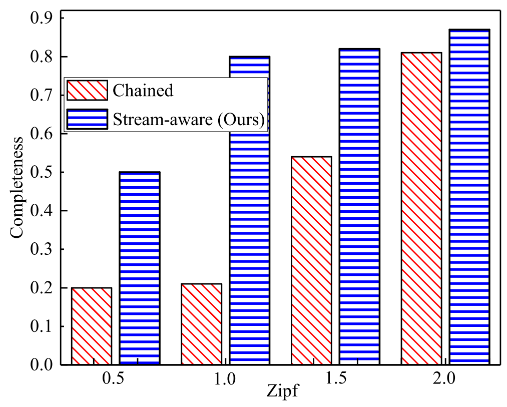

Figure 34: Completeness with 10 PEs and 1sec checkpoint (the higher the better).

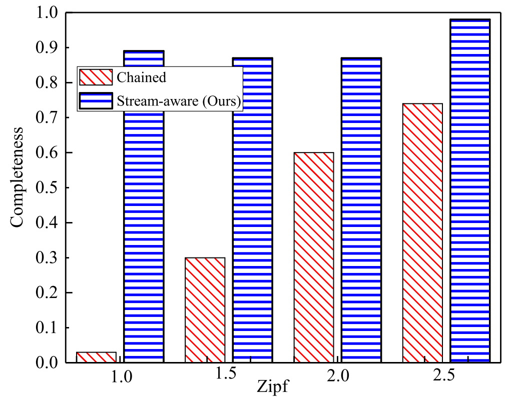

Figure 35: Completeness with 100 PEs and 5sec check pointing (the higher the better).

value the load imbalance among processing instances also increases. However, random data partitioning has a stable imbalance with an increase of Zipf data and has more than 2x times less imbalance than the alternative. Hash-based scheme generate more imbalance because similar keys are processed by the same processing element which generates imbalance for other worker processes.

The proposed scheme uses the checkpoint mechanism for state sharing among PEs where tuples are aggregated and share a common state among them. Figure 33 depicts the average aggregation overhead for round-robin and random partitioning strategy for every 1K tuples against 100PEs. Results depict that every 1k, tuples are aggregated from all underlying PEs whereas, the random scheme uses almost 60 PEs and shows less aggregation overhead than the alternative.

Figure 34 and Figure 35 shows the result for completeness of stream-aware solution with chained index scheme with varying Zipf data. Figure 34 depicts the completeness results with 10 PEs with a checkpoint of 1 seconds. In the case of smaller Zipf=0.5 the chained index is 3.5x time less than the ideal case, where the proposed stream-aware is 1.5x time better than chained index. However, for higher Zipf data the stream-aware solution is approaching to the ideal case. Figure 35 shows the result with 100PEs and the checkpoint interval is increased from 1 second to 5 seconds. The result depicts similar performance characteristics with the previous case with 1.5x to 2x times better performance of the stream-aware solution. Both strategies synchronize their data structures after regular intervals of time. However, the proposed stream-aware maintains an in-memory indexing structure for hotkey elements and only update on change in the frequencies of keys inside this structure. Only non-hot keys data structure share their state among PEs which reduces the chance of the incomplete result set for any on-the-fly search query.

8. Conclusion

Stream inequality join is a fundamental operator for many stream processing applications. Efficient indexing plays an important role in enhancing the performance of stream join solutions. A number of approaches targets this problem with the increasing adoption of DSPSs. When dealing with highly dynamic data, it is more challenging to attain full benefits from the traditional indexing solutions. Thus, we propose a novel indexing solution that considers the popularity of indexing keys and efficiently manages a dedicated in-memory indexing structure for those keys. We call our method STream-Aware inequality join (STA).

STA employes a small size-augmented sketch that effectively maintains the dynamic popularity of the streaming tuples. All identified hot keys are routed towards dedicated processing elements using the power-of-two-choices (POTC). Similarly, cold keys use a random approach for the selection of processing task. The indexes for the sliding window are maintained by these distributed processing elements. We perform a thorough experimental evaluation of the proposed stream-aware inequality join solution against real-world and synthetic data sets on a DSPS (Apache Storm) on a cluster of computing nodes. In the experiments, we run inequality queries for state-of-the-art stream inequality solutions and our STA and measure the throughput, latency, load-imbalance, and completeness. Experimental results prove that using a small-size sketch and our indexing approach helps to improve the performance of stream inequality join predicates in all terms compared to state-of-the-art approaches.

In the future, we aim to extend the inequality join in two distinct dimensions. Firstly, we plan to use a highly efficient immutable structure for the aging keys in the sliding window to expedite the search process. Additionally, we have plans to expand our work to encompass the range partitioning of data within the sliding window. The ranges will be distributed to downstream processing elements, where the non-uniform distribution of data leads to an imbalance in key distribution among the various processing elements. Range repartitioning can help to mitigate this issue, but implementing an efficient dynamic repartitioning technique remains a challenging task. To address this, we are planning to employ a game theoretical approach for adeptly rebalancing the ranges among the worker processes of the distributed streaming processing systems.

References

[1] A. Shahvarani, H.-A. Jacobsen, Distributed stream knn join, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2021, pp. 1597–1609.

[2] A. Michalke, P. M. Grulich, C. Lutz, S. Zeuch, V. Markl, An energy-efficient stream join for the internet of things, in: Proceedings of the 2021 ACM DAMON International Workshop on Data Management on New Hardware, 2021, pp. 1–6.

[3] A. Shahvarani, H.-A. Jacobsen, Parallel index-based stream join on a multicore cpu, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2020, pp. 2523–2537.

[4] V. Cardellini, F. Lo Presti, M. Nardelli, G. R. Russo, Runtime adaptation of data stream processing systems: The state of the art, ACM Computing Surveys 54 (11s) (2022) 1–36.

[5] S. Frischbier, J. Tahir, C. Doblander, A. Hormann, R. Mayer, H.-A. Jacobsen, Detecting trading trends in financial tick data: The DEBS 2022 grand challenge, in: Proceedings of the ACM DEBS International Conference on Distributed and Event-Based Systems, 2022, pp. 132–138.

[6] J. Wu, K.-L. Tan, Y. Zhou, Data-driven memory management for stream join, Information Systems 34 (4-5) (2009) 454–467.

[7] M. Najafi, M. Sadoghi, H.-A. Jacobsen, Scalable multiway stream joins in hardware, IEEE Transactions on Knowledge and Data Engineering 32 (12) (2019) 2438–2452.

[8] Apache Flink, `https://flink.apache.org/poweredby.html`, accessed: 2022-08-03.

[9] Apache Storm, `https://storm.apache.org/Powered-By.html`, accessed: 2022-08-03.

[10] Spark Streaming, `https://spark.apache.org/powered-by.html`, accessed: 2022-08-03.

[11] Z. Xu, H.-A. Jacobsen, Processing proximity relations in road networks, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2010, pp. 243–254.

[12] M. A. Hammad, W. G. Aref, A. K. Elmagarmid, Stream window join: Tracking moving objects in sensor-network databases, in: International Conference on Scientific and Statistical Database Management, 2003, pp. 75–84.

[13] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join, IEEE Transactions on Big Data 7 (2) (2016) 299–312.

[14] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. Ma, V. Markl, Parallelizing intra-window join on multicores: An experimental study, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2021, pp. 2089–2101.

[15] L. Golab, S. Garg, M. T. Özsu, On indexing sliding windows over online data streams, in: Proceedings of the Springer EDBT International Conference on Extending Database Technology, Springer, 2004, pp. 712–729.

[16] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts, Seventh Edition, McGraw-Hill Book Company, 2020.
URL `https://www.db-book.com/`

[17] Q. Lin, B. C. Ooi, Z. Wang, C. Yu, Scalable distributed stream join processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2015, pp. 811–825.

[18] F. Pan, H.-A. Jacobsen, Panjoin: A partition-based adaptive stream join, arXiv preprint arXiv:1811.05065 (2018).

[19] M. Najafi, M. Sadoghi, H.-A. Jacobsen, {SplitJoin}: A scalable, low-latency stream join architecture with adjustable ordering precision, in: Proceedings of the USENIX ATC Annual Technical Conference, 2016, pp. 493–505.

[20] P. Roy, J. Teubner, R. Gemulla, Low-latency handshake join, Proceedings of the VLDB Endowment 7 (9) (2014) 709–720.

[21] B. Gedik, R. R. Bordawekar, P. S. Yu, Celljoin: a parallel stream join operator for the cell processor, The VLDB journal 18 (2) (2009) 501–519.

[22] S. Zhou, F. Zhang, H. Chen, H. Jin, B. B. Zhou, Fastjoin: A skewness-aware distributed stream join system, in: Proceedings of the IEEE IPDPS International Parallel and Distributed Processing Symposium, 2019, pp. 1042–1052.

[23] P. Roy, A. Khan, G. Alonso, Augmented sketch: Faster and more accurate stream processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2016, pp. 1449–1463.

[24] N. Manerikar, T. Palpanas, Frequent items in streaming data: An experimental evaluation of the state-of-the-art, Data & Knowledge Engineering 68 (4) (2009) 415–430.

[25] G. Cormode, S. Muthukrishnan, Summarizing and mining skewed data streams, in: Proceedings of the SIAM International Conference on Data Mining, 2005, pp. 44–55.

[26] T. Kriechbaumer, H.-A. Jacobsen, BLOND, a building-level office environment dataset of typical electrical appliances, Scientific Data 5 (1) (2018) 1–14.

[27] Z. Jerzak, H. Ziekow, The DEBS 2015 grand challenge, DEBS '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 266–268. `doi:10.1145/2675743.2772598`.
URL `https://doi.org/10.1145/2675743.2772598`

[28] O. Rottenstreich, Y. Kanizo, I. Keslassy, The variable-increment counting bloom filter, IEEE/ACM Transactions on Networking 22 (4) (2013) 1092–1105.

[29] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, Journal of Algorithms 55 (1) (2005) 58–75.

[30] Q. Zhang, F. Li, K. Yi, Finding frequent items in probabilistic data, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008, pp. 819–832.

[31] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, S. Uhlig, Cold filter: A meta-framework for faster and more accurate stream processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2018, pp. 741–756.

[32] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, M. Serafini, The power of both choices: Practical load balancing for distributed stream processing engines, in: Proceedings of the IEEE ICDE International Conference on Data Engineering, 2015, pp. 137–148.

[33] J. Kang, J. F. Naughton, S. D. Viglas, Evaluating window joins over unbounded streams, in: Proceedings of the IEEE ICDE International Conference on Data Engineering, 2003, pp. 341–352.

[34] J. Teubner, R. Mueller, How soccer players would do stream joins, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2011, pp. 625–636.

[35] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, P. Kalnis, Lightning fast and space efficient inequality joins, Proc. VLDB Endow. 8 (13) (2015) 2074–2085.

[36] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al., Synopses for massive data: Samples, histograms, wavelets, sketches, Foundations and Trends® in Databases 4 (1–3) (2011) 1–294.

[37] M. Ahmed, Data summarization: a survey, Knowledge and Information Systems 58 (2) (2019) 249–273.

[38] I. Mytilinis, D. Tsoumakos, N. Koziris, Workload-aware wavelet synopses for sliding window aggregates, Distributed and Parallel Databases 39 (2) (2021) 445–482.

[39] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in: Proceedings of the Springer ICALP International Colloquium on Automata, Languages, and Programming, 2002, pp. 693–703.

[40] Y. Zhao, Y. Zhang, P. Yi, T. Yang, B. Cui, S. Uhlig, The stair sketch: Bringing more clarity to memorize recent events, in: Proceedings of the IEEE ICDE International Conference on Data Engineering, 2022, pp. 164–177.

[41] P. Jia, P. Wang, J. Zhao, Y. Yuan, J. Tao, X. Guan, Loglog filter: Filtering cold items within a large range over high speed data streams, in: Proceedings of the IEEE ICDE International Conference on Data Engineering, 2021, pp. 804–815.

[42] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., Storm@ twitter, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.

[43] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, M. Serafini, When two choices are not enough: Balancing at scale in distributed stream processing, in: Proceedings of the IEEE ICDE International Conference on Data Engineering, 2016, pp. 589–600.

[44] H. Chen, F. Zhang, H. Jin, Popularity-aware differentiated distributed stream processing on skewed streams, in: Proceedings of the IEEE ICNP International Conference on Network Protocols, 2017, pp. 1–10.

[45] M. Adler, S. Chakrabarti, M. Mitzenmacher, L. Rasmussen, Parallel randomized load balancing, in: Proceedings of the ACM STOC Symposium on Theory of Computing, 1995, pp. 238–247.

[46] M. Mitzenmacher, On the analysis of randomized load balancing schemes, in: Proceedings of the ACM SPAA Symposium on Parallel Algorithms and Architectures, 1997, pp. 292–301.

[47] J. Yuan, Y. Wang, H. Chen, H. Jin, H. Liu, Eunomia: Efficiently eliminating abnormal results in distributed stream join systems, in: Proceedings of the IEEE/ACM IWQOS International Symposium on Quality of Service, 2021, pp. 1–11.