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 methods and optimizations exist for efficient inequality join in batch processing, little attention has been given to its streaming version, particularly to large-scale data-intensive applications that run on *Distributed Stream Processing Systems* (DSPSs). Designing an inequality join in streaming and distributed settings 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 four state-of-the-art inequality join solutions for a widely employed DSPS (Apache Storm) and compare their performance with STA on four real-world data sets and a synthetic one. The results of our experimental evaluation reveal that our

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

stream-aware approach outperforms existing solutions.

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 both industries and the scientific community. Streaming applications include, for instance, real-time analysis of social networks [1], smart grid management [2], fraud detection [3], network intrusion identification [4], and online financial trend indicators [5, 6, 7]. All of these applications require meaningful insights into a continuous flow of data with low latency and high record processing throughput.

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, including tweet analysis, searching, and revenue optimization [9], and Freeman Lab at Howard Hughes Medical Institute[5] uses Apache Spark Streaming[6] for real-time analysis of human brain actions [10]. A DSPS (such as the aforementioned 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 real-world scenarios, such as recommendation systems, online car ride and sharing applications, online monitoring and analysis of multi-sensor data [1, 11, 12, 13].

[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

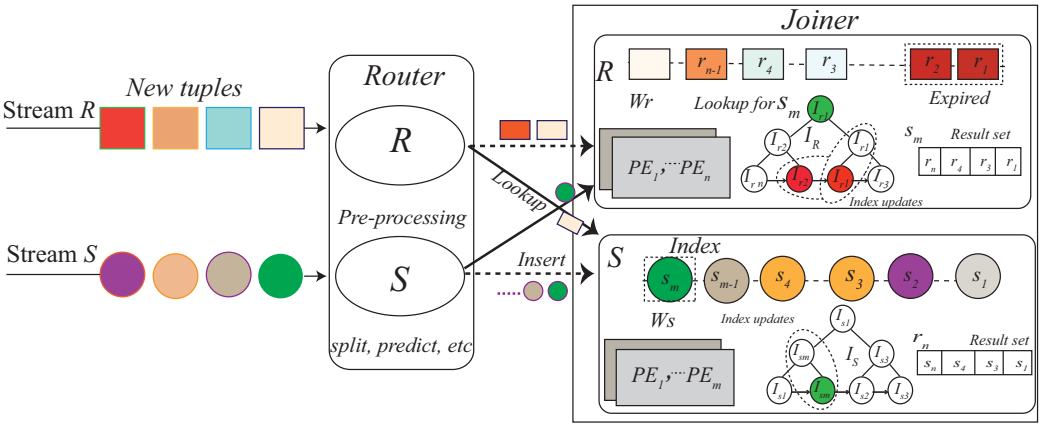


Figure 1: Distributed window-based stream join processing

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 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 several tuples (i.e., a window) as depicted by Figure 1. A join performed in this way is called *stream window join* or *window-based stream join* [3, 6, 14], denoted for short as *windowing* in this paper when evident from the context. When the join also requires to perform inequality comparisons (i.e., using operators such as "$<$", "$>$", "$\leq$", or "$\geq$"), it is called *inequality join*. This kind of join is widely employed in many scenarios and raises several challenges, as we discuss in this paper. In particular, the following example provides an intuition of the usefulness of this operator in real-world use cases.

Example 1.1. *A smart grid analyst monitors real-time power consumption data from remote smart meters in a smart grid system with two power generators, R.Gen (lower capacity) and S.Gen (higher capacity). He instantiates a query to detect if R.Gen power consumption (R.Power) exceeds S.Gen power consumption (S.Power) for a sliding window of 60 minutes with a slide interval of 10 minutes. Identifying such an imbalance helps the analyst promptly shift the load by reducing demand on R.Gen and increasing it on S.Gen.*

Q 1: Query for real-time power consumption analysis for smart grid.

```
SELECT R.POWER, S.POWER, Timestamps
FROM R.GEN JOIN S.GEN
    ON R.POWER > S.POWER
WINDOW w AS SLIDE '10' MINUTES AND '60' MINUTES LENGTH
```

Further, consider another example with a *band join*, which contains an inequality operator that returns all pairs of tuples that are close to each other [15].

Example 1.2. *The network administrator of an ISP monitors network traffic in real-time to make informed decisions. They use a real-time, time-based windowing query to evaluate traffic flows between two key routers, R.Router and S.Router. This query compares the size of data packets and identifies instances where the absolute difference between R.Data_Size and S.Data_Size is below a predefined threshold within a specified time for each packet. The administrator can effectively redistribute the load by analyzing these filtered packets, ensuring balanced network operation.*

Q 2: Query for real-time Internet traces analysis.

```
SELECT R.DATA_SIZE, S.DATA_SIZE, Timestamps
FROM R.ROUTER JOIN S.ROUTER
    ON ABS(R.DATA_SIZE-S.DATA_SIZE)<Threshold
WINDOW w AS SLIDE 'd' MINUTES AND 's' MINUTES LENGTH
```

The examples mentioned above involve inequality join operators between two opposite streams R and S as depicted by Q 1 and Q 2. Additionally, multiple sources generate data for these streams simultaneously. To efficiently analyze such streaming data in real-time, we utilize DSPSs, which can effectively scale out the computation for improved performance. The state-of-the-art approach for distributed join processing follows a *router* and *joiner* model, as depicted in Figure 1. The application is submitted to the DSPS as a Directed Acyclic Graph (DAG). The DAG contains two key components in this scenario: *router* and *joiner*. Consumers of the DSPS consume input streams from multiple sources through the *router*, as depicted in Figure 1 [16]. This component also performs some pre-processing on each input streaming tuple, which includes splitting or some data cleaning operations. Furthermore, in the initial computation, such as in our study, we use a key

frequency predictor module, which is also performed in the *router* component. The *router* then sends the pre-computed tuple downstream to the joiner component. The *joiner* contains multiple PEs that hold the tuples for the streaming window. Additionally, the stream join operation is also performed in this component.

Consider example 1.1 as a use case. A new tuple from stream R is firstly inserted into the *router* component. After the initial computation, the tuple is sent downstream to both nodes (R and S) of the *joiner*. The new tuple from stream R is inserted into the R node. Simultaneously, this tuple performs a lookup operation on the node S. Finally, the results are transformed into the new node of the DAG for further processing. The key operations among processing elements of each node include efficient index updating, simultaneous lookup operation among distributed processing elements (PEs), data partitioning among PEs of joiner component, and state management for streaming windows.

Hash tables are commonly employed for processing queries with equality operator due to their efficiency in inserting and searching for tuples [3, 17]. However, for non-equality predicates, hashing is inefficient as it requires too many memory scans. In traditional batch DBMSs, indexing is a better choice for maintaining the logical order of content [3, 18]. However, the nature of streaming data differs from 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 operations are frequent for online data, creating an index restructuring overhead for highly dynamic data. In Figure 1, I_R and I_S represent the index data structures for sliding windows of streams R and S, respectively. A new tuple s_m from stream S looks at I_R for predicate evaluation and is indexed into I_S for future tuple evaluations. 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 it 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 [16], partitioned in-memory tree [3], Parallel-MJ [7], Panjoin [19], split-join [20], low latency handshake join [21], cell join [22]. The main intuition of these schemes is to use sub-indexes for a single sliding window that exists independently on each core of a single processor or *processing elements* of a DSPS. These solutions eliminate the update and deletion cost to an extent

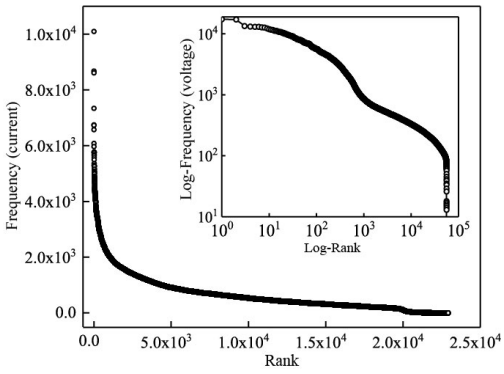


Figure 2: Frequency distribution of current and voltage data for the BLOND data set.

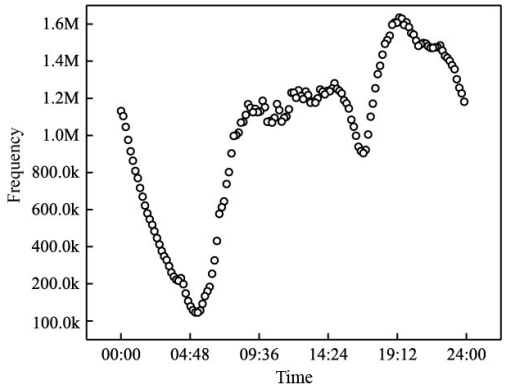


Figure 3: Frequency distribution of NYC taxi requests w.r.t. time.

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 others. For example, mobile transportation [23], IP traces [24], clickstream data [25], sensor readings, email and SMS logs all follow the Zipf distribution [26]. In particular, Figure 2 shows the data distribution of current and voltage for the Building Level Office environment Data set (BLOND) [27], while Figure 3 shows the frequency distribution of taxi requests for the New York City (NYC) taxi data set [28].

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 4 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 5 shows that the use of separate indexing reduces the redundancy of top-k elements in the sub-indexes of the streaming window.

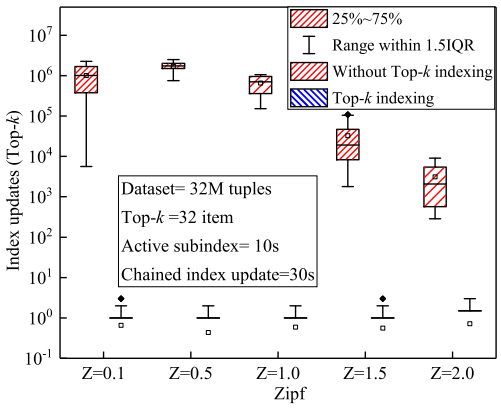


Figure 4: Index update for with and without top-k elements

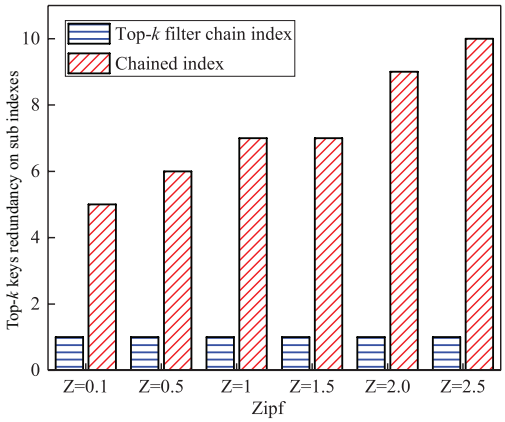


Figure 5: Redundancy of keys in sub-indexes

Several approaches have been proposed for stream data approximation including hash-based or key-based schemes: hash-based schemes such as counting Bloom filter [29] and count-min sketch (CMS) [30] 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 [31], 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 especially for streaming data. [32].

1.2. Contribution

In our research, we present a solution for dealing with distributed inequality by considering the distribution of the input tuples for indexing data items. Newly added tuples are partitioned among the processing elements of the joiner component based on the frequency of the key. We make use of the power-of-the-two-choices (POTC) method for indexing hotkeys [33]. These hotkeys are actively monitored by an efficient sketch in the router component of the join processing. The sketch continuously keeps track of the hotkeys. Using power-of-two-choices is seen as a better alternative to a single hash choice, as it helps to prevent load imbalance caused by frequently used keys. We steer clear of using more choices for hotkeys as it would add extra cost by requiring a reduction node to aggregate partial results from the upstream

processing elements.

For cold items, we use a random data partitioning strategy. Each processor holds a local indexing data structure for indexing. This structure is continually synchronized with all processing elements of cold items at regular checkpoint intervals. Additionally, a non-frequent key from the hot indexing data structure is also inserted randomly into the processing elements of cold keys. This approach helps to avoid high index construction costs on processing elements. Furthermore, new tuples do not require lookup operation to all processing elements for the completion of query results. A new tuple is broadcast to the two dedicated PEs of hotkeys and one PEs which contains the chain of the non-hot keys index structure. We face several significant challenges when addressing inequality in DSPS, as outlined in Section 2.4.

The overall contributions of this work are:

- We propose a new indexing strategy based on key frequency for the inequality join operator for streaming data. A new tuple requires only a few processing elements to complete the query results.
- 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 hotkey 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 is organized as follows: Section 2 provides a comprehensive overview of existing work on stream window join algorithms, data partitioning schemes, preliminaries, cost analysis of state-of-the-art approaches, and challenges associated with the proposed study. Section 3 describes the proposed stream-aware solution. Section 4 provides an implementation detailed of inequality join algorithms for DSPSs and the experimental setup for eval-

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

uation, while Section 5 offers insights into the experimental evaluation and provides discussion. Finally, Section 6 concludes the paper.

2. Related Work and Preliminaries

We first provide the related work for the stream join solution (Section 2.1). Secondly, we provide preliminary details for stream inequality join (Section 2.2). Thirdly, we provide cost analysis for the state-of-the-art inequality join strategies (Section 2.3). Finally, we provide a discussion on challenges (Section 2.4) associated with the implementation of STA-Join and other join strategies in DSPS.

2.1. Related work

In this subsection, we provide related work about the stream window join, state-of-the-art top-k prediction techniques, and data distribution among worker processes of DSPSs

2.1.1. Stream window join

Kang et al. [34] provides 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.

A study by Teubner et al. [35] provides a solution for such hardware that utilizes a high level of parallelism for stream window join computation that is named handshake join (HSJ). 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's handshake, and evaluation result $\langle r, s\rangle$ is added into the result set. Roy et al. [21] 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. [21] proposes a low-latency handshake join (LLHJ) where the contents of the streaming window searched from all available cores of the system. As analogous to handshake join, tuples of individual streams 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 [20]. Moreover, the tuples are deleted individually from each core that reshuffling the indexing structure of core c_i. Single tuple removal decreases the processing throughput and increases the concurrency overhead.

Split join [20] 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 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 [20] 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. [16] introduces a novel model for stream join processing in DSPS named 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 for new incoming data. However, with the increased number of chained indexes, a new tuple requires lookup operation on 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] proposes a new design for indexing stream items of the window that consists of mutable and immutable data structures. 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 queries, 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 reconstructions of new tuples.

A study by Pan et al. [19] proposes 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, preprocesses the data, and distributes the tuples toward the processing node for predicate evaluation. 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. [36] proposes 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.

2.1.2. Multi-dimensional index structures

Multi-dimensional index structures are commonly employed for spatial data and are also known as spatial indexes [37]. The spatial indexes can be broadly categorized into two types: tree-based or grid indexes [38]. Tree-based indexes are typically used for static batches of spatial data. However, they are not ideal for real-time data with continuous insertion and deletion, such as R-Tree [39]. To address this challenge, grid indexes are used to first define the data boundaries. The insertion or removal of tuples within the grid requires less maintenance. Recent grid indexes used for spatial data indexing include GeoFlink [38], SQUID [40], GHOST [41], and comprehensive review by Li et al [37] that focuses on different techniques used for multi-dimensional indexes for range queries and k-nearest neighbor queries (kNN). While these grid and multi-dimensional indexes are optimized for spatial data, our proposed approach focuses solely on non-spatial data. However, these data structures can also be used for the proposed distributed join data structures for multi-dimensional data.

2.1.3. Stream data prediction

It is impractical to save and then manage the huge volume of continued data. Synopsis plays an important role in the analysis and prediction of continuing data. Cormode et al. [43] provides a comprehensive description of synopsis construction that includes: sampling (random sampling, stratified sampling, chain sampling, priority sampling, etc) [44], wavelets [45], sketches (count-bloom filter, count-min sketch, and cold-filter) [29, 30, 32], and top-k estimators (space-saving) [26]. Sketches are normally used to approximate the frequency of data stream elements. Several studies consider these sketches for top-k identification of streaming data [46, 43], however, the top-k estimators (e.g, space-saving [43]) provide a more accurate and efficient top-k return with high throughput [24] algorithm. Sketches need a large number of hash functions and high sketch size for higher accuracy,

whereas, in a space-saving algorithm there is a frequent exchange of stream that can degrade top-k prediction accuracy.

Several studies propose novel solutions by augmenting these state-of-the-art filters, sketches, and counter-based approaches (e.g., cold filter [32], stair sketch [47], loglog filter [48], augmented sketch [24]). The main theme behind these approaches is to separate the hot and cold items from the continuous data. Cold filter [32] consists of three layers, where the 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 [47] mainly focuses on the recent event stream and its accuracy instead of all items in the data stream. The staircase sketch considers the time stamps and organizes the memory structure accordingly. It uses the basic bloom filter and CMS for testing the approach. Loglog filter [48] complements the cold filter [32] and filters the cold items of the stream, where the filter uses less memory space and holds more items than the cold filter.

Table 1: Classification of related work to STA-Join

Techniques	**Type**	**Shortcomings**	**STA-advantages**
HSJ [35], LLHJ [21], SJ [20]	Simple window join	Tuple removal and state management are challenging and require exploration of all cores or PEs for completeness.	The strategy involves removing coarse-granular tuples. The new tuple requires fewer processing elements and exploration for completion.
CI [16], PIM [3], PAN-Join [19], IE-Join [36]	Indexing schemes	All methods aim to address inequality. A tuple involves the simultaneous use of various index structures.	It only utilizes a single index-type structure in parallel on a limited number of processing elements.
BCHJ, Fast Join [23], Simos [42]	Partitioning-based	Typically utilized for equality join operations, these techniques often aim to mitigate load imbalance issues among distributed worker processes.	Load imbalance arises from 'hotkeys,' prompting equal workload distribution across two partitions.
GeoFlink [38], SQUID [40], GHOST [41]	Multi-dimensional indexing	Commonly used for spatial data, but it is expensive for non-spatial high-order range queries.	Optimizes for non-spatial high-order ranges. It can be employed for spatial indexes.

Augmented Sketch [24] filters hot items of the stream by using a two-layer data structure. The first layer keeps a key-value pair for item frequency, whereas the second layer uses the CMS for approximation. Moreover, augmented sketch dynamically handles the popularity of the keys by swapping the items between layers depending on their frequency. In this work, we use the augmented sketch to keep track of hot items in the streaming element.

2.1.4. Data distribution among worker processes of DSPS

In DSPS the processing is performed on a multi-node cluster. Several 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 nodes. These partitioning strategies selection depends on the data and business logic. The shuffle grouping [49] uses a round-robin scheme for partitioning the data among distributed nodes. It is better suited for stateless operations; however, in stateful data processing, data items require an extra aggregation cost. Key grouping [49] 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 [33] uses two hash functions for a single data item to reduce the impact of load imbalance among processing instances.

Identifying hotkeys in a stream and partitioning them into more than two processing elements can reduce the load imbalance among instances of the data element. Nasir et al. [50] propose a hotkey-aware data partitioning solution. They use counter-basis space-saving data structures to keep track of top-k elements and route these elements to more than two processing elements using w-choices and d-choices, while cold items are treated by using power-of-two-choices. Chen et al. [51] propose a novel counting-based probabilistic solution for the dynamic tracking of hotkeys. All hotkeys are processed by all worker processes in a round-robin fashion, whereas cold items use hash-based field grouping. This approach helps to reduce the load imbalances among worker processes; however, creates a huge overhead for tuple accumulation from multiple processing instances for stateful computation. Zhang et al. [23] 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 an exponential counting mechanism that keeps records for heavy hitters in the stream and uses shuffling for frequent items and hashing for non-frequent keys in the stream. In our

proposed study, we use POTC that partitions the high frequency for separate indexing to balance the load among workers and use random choice for all other keys [52].

2.2. Preliminaries

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

2.2.1. Stream join processing

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

Definition 2.1. *(Streaming data): is a continues flow of infinite data items known as tuples* $T = \{t_1, t_2, t_3 \dots\}$. *Each tuple* t_i *consist of finite set of keys and values pair* $V = \{\langle k_1, v_1\rangle, \langle k_2, v_2\rangle, \langle k_3, v_3\rangle, \dots, \langle k_n, v_n\rangle\}$, *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 2.2. *(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* $W_d = \{t_d, t_{d-1}, t_{d-2}, \dots, t_{d-n}\}$ *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 2.3. *(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 consider 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 2.4. *(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 [16] 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*.

2.2.2. Augmented sketch

Augmented sketch (ASketch) [24] 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. [30]. 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 $S_o = \{\exists k_i \mid (k_i \in K), minH(k_i) > L_1^{k_{top}}\}$.

2.2.3. 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. [49] explains the semantics of data processing of a DSPS in 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 [49], partial key grouping [33], key grouping [49], 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 creates 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

Table 2: 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^w_{di}	Expire tuple
t^{max}_{kr}, t^{min}_{kr}	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

for the completeness of the result. Hashing the particular key and selecting the processing task based 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.

2.3. Stream join solutions for DSPS

In this subsection, we provide the cost analysis 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 several processing tasks that operate in a parallel fashion. The notations used are explained in Table 2.

2.3.1. Broadcast join

Broadcast join is widely used in Spark batch processing join [10]: the smaller side of the data set is broadcasted to all executors of the cluster nodes, then the join is performed on the candidates of the 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 datasets this scheme creates a serious memory overhead, moreover, for non-equi join it increases the latency while completing the θ evaluation.

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 a key k_i from an array of PEs, broadcast hash join has the same cost as 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. (4).

$$C_{BCJ}^{k_i} = 2n(c_i^{pe}) + H(k_i) + \sum_{i=1}^{n} W_i^{PE} \tag{4}$$

2.3.2. Split join

Split join [20] divides cores of a processor for dedicated streaming windows W_R and W_S. A stream item k_i needs to search all cores. Following the semantics of the multi-core system, we use processing tasks instead of cores. Each PE holds 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.

The cost of the insertion includes finding the PE and insertion of the new tuple to that identified PE. 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 costs of exploring the local windows of PEs, so the total cost for a split join can be described by Eq. (5).

$$C_{Split-join}^{k_i} = (x + 2(c_i^{pe})) + \sum_{i=1}^{n} PE_i^w \tag{5}$$

2.3.3. Chained-index

Chain-index [16] is implemented for Apache Storm, however, its implementation details are missing from the description [16]. 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.

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 check-pointing factor also affects the total cost of the chained index. The cost can be represented by Eq. (6) where ρ is the cost for checkpoints that depend on time.

$$C^{k_i}_{Chained-index} = (H(k_i) + c_y) + \sum_{i=1}^{n} y^r_i + \rho_{time} \tag{6}$$

2.3.4. PIM-Tree

The partitioned in-memory merge tree is designed for multi-core systems. It is a two-layered data structure I^1_p and I^2_p 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.

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. (7).

$$C^{k_i}_{PIM-tree} = (H(k_i) + c_y) + (\sum_{i=1}^{n} (y^r_i) + c_I^{Immutable}) + M + \rho_{time} \tag{7}$$

2.4. Challenges

In this subsection, we consider the challenges associated with STA-Join and other join implementations in DSPS.

2.4.1. Challenges with STA-Join

We deal with several significant challenges when working with STA-Join for DSPS. Real-time identification of the frequent keys is a challenging task. Although we have implemented ASketch [24] for real-time identification and updating of frequent and non-frequent keys, however, we need an efficient mechanism for updating the indexing data structures that store hot and non-hot keys in sliding windows. Selecting the appropriate data partitioning strategy for hot and non-hot keys is also a challenging task. Key partitioning may cause load imbalance, while round-robin partitioning introduces extra overhead for aggregating tuples from various processing elements for the result set. To address this, we have chosen power-of-two-choices for hotkeys to reduce downstream instances load due to the payload of hotkeys, and a random approach for non-hot keys. The less frequent keys maintain a common chained index of input tuples, however, maintaining and updating this chained index to ensure a consistent state among all processing elements is difficult.

2.4.2. Challenges with state-of-the-art approaches

This study also considers the state-of-the-art approaches to handling inequality joins in stream processing systems. Existing approaches such as Split Join [20] and PIM-Join [3] are specifically designed for multi-core systems. Adapting DSPS processing semantics to these algorithms is a significant and challenging task. Similarly, another approach, Chain Index [16], does not explain how a sliding window maintains and updates a consistent state while its items are held by multiple Processing Elements (PEs) of joiner components. We have implemented a checkpoint-based state synchronization mechanism. However, there is still a trade-off between the synchronizing interval of state among PEs and the completeness of results.

3. Stream-aware Join Processing

In this section, we provide details about architecture for stream-aware join processing that includes: the use of stream predictor (Section 3.1) for data partitioning (Section 3.2), data distribution among processing nodes, use of separate indexing for hot and cold items of streaming data (Section 3.3), stream join (Section 3.4), and dynamic change in popularity of key during stream processing. The architecture of the proposed stream-aware index solution is shown in Figure 6.

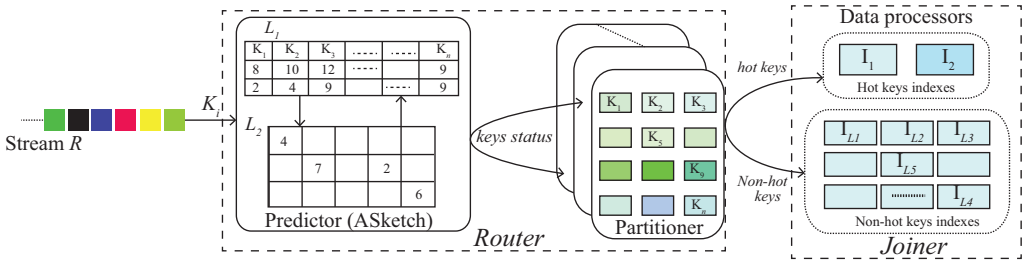


Figure 6: Architecture for stream-aware indexing

3.1. Stream predictor

We employ ASketch [24] to keep track of data popularity. As discussed in Section 2.2 the ASketch 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 model a special vertex of DAG is used to build ASketch in the *router* part. Every key of the streaming data enters into the ASketch as depicted by the predictor of Figure 6. 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 15 explains how we take advantage of ASketch for keeping track of hot and cold keys in streaming data.

The input to Algorithm 15 is the streaming data, fixed size of the filter, and CMS size. Initially, a key k_i enters into L_1 of ASketch and then linearly scans the filter to check the presence of k_i in L_1 as depicted by line 1 of Algorithm 15. 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 15. However, in a case where the filter has not enough space to accommodate the new k_i then the key 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 15. 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 15.

Algorithm 1: Tuples prediction with ASketch

Input: Key (k_i), L_1 size (ϕ), and L_2 (HxW)
Output: k_i popularity

```
Scan the layer L1 linearly;
if k_i ∈ L1 then
    update N_c to N_c + 1;
    return hot(k_i);
else if | k_i | ≤ φ then
    insert k_i in L1;
    return non-hot(k_i);
else
    insert k_i into CMS;
    if min Hf(k_i) > L1^{k_Top} then
        update L1^{k_Top} with min Hf(k_i);
        insert L1^{k_Top} frequency N_c − O_c in L2;
        return hot(k_i);
    else
        return non-hot(k_i);
```

3.2. 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 nodes of *joiner*, as shown in Figure 6. 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. [33] explains that using two choices instead of one improves exponential improvement for load balance than a 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 5. 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

Algorithm 2: Data partitioning for join processing

Input: PEs, Algorithm15 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) ← 0;
3      current (c_j) ← 0;
4      choices C = {PE_{n-1}, PE_{n-2}} ;        // PEs can be added in C
5      while True do
6          c_i ← c_j + 1;
7          if c_i<|C| then
8              return C[c_i];
9          else if c_i==|C| then
10             c_j ← 0;
11             return C[PE_{n-1}];
12 else
13     available choices C = {PE_1, PE_2, PE_3, ... PE_{n-3}};
14     return k_i ← rand{C}
```

stream are routed with a randomized approach that follows the well-known *super market* model [52, 53]. 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 [52]. 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 5.

Algorithm 14 provides the complete procedure for data partitioning for stream data windowing in the *joiner* component. The input to the algorithm is available processing tasks and the output of Algorithm 15 $\langle k_i, K_i status \rangle$, where $K_i status$ indicate the key popularity in the stream. The output of the Algorithm 14 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 14. 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

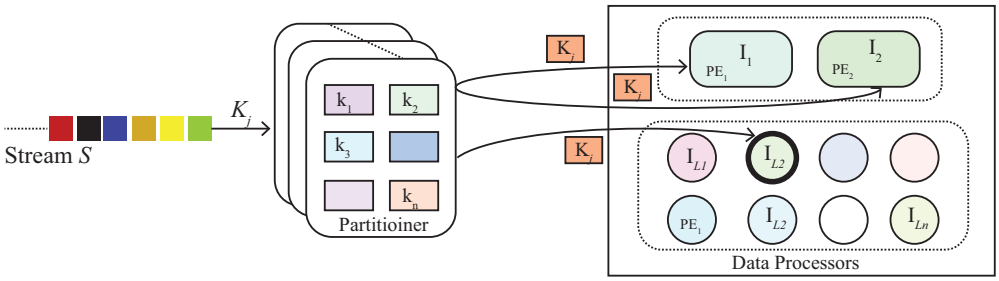


Figure 7: Data partitioner for stream join

allocated to the first. This whole assignment procedure is described by lines 5 to 11 of Algorithm 14. 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 14. 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 7. 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 for non-frequent keys. 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.

3.3. Sliding window indexing

Queries that involve non-equality predicates require an efficient indexing structure to accelerate the query processing, however, it creates the overhead 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 6. 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 8. However, using

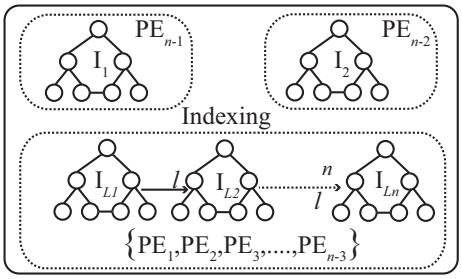


Figure 8: Indexing window contents among PEs

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 check-pointing and can return an incomplete result, however, we measure the system performance against synthetic increasing skewed data in Section 5, 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-indexes using a linked list as shown in Figure 8. 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 [16, 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 a non-hot key at any instance becomes popular during flow of the 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 reduces the rate of redundancy on sub-indexes for increasing Zipf as depicted by Figure 5.

- If hotkey frequency is changed to un-popular during stream processing then the key is removed from the indexing structures of hotkey 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. [24] explains increasing *zipf* has a little tuple swapping overhead.

3.4. 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 speeds up the lookup procedure. Our proposed indexing structure is shown in Figure 8. Moreover, stream joins 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 adds 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 hotkeys 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-hots 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 the sub-tree 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 depends 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 [16] 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. (8).

$$C_{lookup} = I_1 + I_2 + \sum_{i=1}^{n} I_{Li} \tag{8}$$

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.(9). 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.

$$C_{update} = \begin{cases} 1/2, & \text{if } k_i \text{ is hot key} \\ 1/n, & \text{otherwise, n is total sub-trees in } \mathrm{I}_{Li}. \end{cases} \tag{9}$$

Expired tuples from the sliding window are removed during the insertion of the 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} [16, 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 by using ASketch and show the superior performance of the stream join algorithm on high selective queries in Section 5.

4. Experimental Setup

In this section, we first map the semantics of exiting stream-join solutions by considering the DSPS operators (Section 4.1). Secondly, we describe the configuration of the cluster on which we run our experiments (Section 4.2); then, the different data sets (Section 4.3) and finally, the metrics (Section 4.4) used for the evaluation of our novel solution against current state-of-the-art baselines.

4.1. State-of-the-Art stream inequality join implementation

In this subsection, we provide an implementation detail for the state-of-the-art join strategies in DSPS as explained by Algorithm 3.

4.1.1. Broadcast hash join

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.

4.1.2. Split 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

Algorithm 3: Stream join algorithms for DSPS

Input: $\{PEs\}$, k_i, W, θ condition
Output: $k \bowtie_\theta W$

```
1  switch Strategy do
2      case BCHJ do
3          Insertion: k_i ↦ ∀{PEs} ;                    // same W on each PE
4          Look up: PE_1 ← H(k_i) ;                     // H to select PE for θ
           evaluation
5          Remove: ∀{PE_i^W}\k_i ;   // Remove k_i depends on threshold
6      case Split-join do
7          Insertion: k_i ↦ {pe ∈ {PE}| pe is selected round robinly};
8          Look up: k_iθ → ∀{PEs};
9          Remove: (k_i ∈ PE)\k_i
10     case Chained-Index do
           // shared a linked sub index among all PEs
11         Insertion: k_i ↦ {y_s ∈ {I_c}| y is sub-index of I_c };
12         Look up: k_iθ → ∀y_i^n : i ∈ {1 to n};
13         Remove: I_c\y_s: y_s > λ ;                   // λ is threshold
14     case PIM-Tree do
           // shared a linked sub index among all PEs
15         Insertion: k_i ↦ {y_s^r ∈ {I_c^r}| y^r is ranged sub-index of I_c^r};
           Look up: k_iθ → ∀y_i^r : i ∈ {1 to n} ∪ I_merged;
16         Remove: // coarse-grained tuple removal
17         if lim |I_c^r| > ω then
               // ω is threshold for merging
18             Merge ∀y_s^r to I_merged;
19             ∀k_{i→n} > γ \{I_c^r, I_merged}; // γ is threshold for removal
20
21     otherwise do
22         exit
```

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

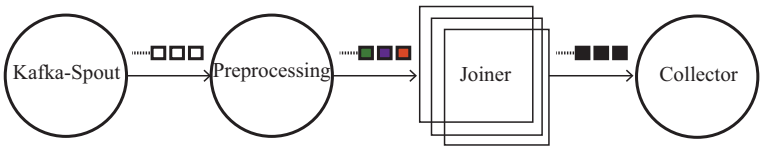


Figure 9: Topology view for join processing.

during insertion depends on a certain threshold.

4.1.3. Chained index

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 check-pointing. 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, coarse-grained tuples are removed, where the whole sub-index is removed from the linked list.

4.1.4. PIM-tree

In DSPS, we use a similar methodology of the chained index for mutable components, by using a single hash function for tuple routing among processing tasks and 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 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.

4.2. Cluster setup and topology

All inequality join algorithms are implemented on the top of Apache Storm. The cluster consists of 10 distributed machines with varying computing resources, with RAM ranging from 4 to 12 GB and disks from 120 to 500 GB. For state management, especially for the proposed STA-Join solution, we adopt Redis as a distributed in-memory cache. Additionally, we use Apache Kafka 3.5.0 to ingest data into the Storm cluster. Apache Zookeeper is employed for coordinating Storm and Kafka cluster nodes. In particular, one node serves as the *Nimbus* master service, while the other nodes act as *Supervisor* client services. Moreover, for time synchronization, we employ NTP server, on the *Nimbus* node.

Table 3: Datasets and inequality query types

Queries	Dataset	Join type	Zipf	Size
Q 3	NYC taxi	Self join	2.24	173M
Q 4	BLOND	Cross join	1.8	200M
Q 5	Internet traces	Cross band join	1.4	14M
Q 6	Stock market	Self band join	1.22	10M
Q 7	Zipf	Self & cross	1.0- 2.0	32M

In Figure 9, the Storm cluster uses a Kafka Spout to consume streaming data, which is then sent to the *preprocessing* bolt. At the preprocessing stage, certain tasks are performed on the streaming tuple, such as the key popularity identification algorithm for the proposed STA-join. Additionally, the choice of partitioning strategy, which depends on the join algorithm, is also determined at this stage. After preprocessing, the tuples are sent downstream to the joiner component for sliding windows and join computation. Finally, the results are collected at the *collector* node of the inequality join DAG.

4.3. Data sets and predicate description

We used four actual data sets and synthesized data sets with varying levels of *Zipf* for our evaluation. We chose different queries based on the size and characteristics of the dataset. Additionally, for each query, the sliding interval (D) and window length (W) varied for each experiment. The dataset and queries description are summarized in Table 3.

- The NYC taxi data set, from the DEBS grand challenge 2015 [28]. This data set contains information regarding taxi trips with 17 various fields and 173 million events. We consider the $fare$ as a metric for indexing and evaluate a single predicate with a self-join as depicted by Query 3.

Q 3: Query for NYC Taxi data.

```
SELECT *
FROM NYCTaxi T1 JOIN NYCTaxi T2
    ON T1.FARE > T2.FARE
WINDOW w AS (SLIDE INTERVAL 'D' ON 'W');
```

- The BLOND data set contains the continuous measurement of voltage V and current I of aggregated circuit inside an office measurement

[27]. Run time analysis can help with efficient load profiling, power saving, and automation. We use a 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 The total size of the tuples is 200M. We use the product of V and I as a power consumption of generators in a smart grid infrastructure. This query Q 4 can be useful for real-time scheduling of load to different power generators.

Q 4: Query from BLOND data set (cross join)

```
SELECT *
FROM R.GENERATOR JOIN S.GENERATOR
    ON R.POWER > S.POWER
WINDOW w AS (SLIDE INTERVAL 'D' ON 'W' )
```

- Internet traces dataset consists of packet-level network traces categorized into five different types: bulk, video, web, interactive, and idle. [8] We simulated the size of these network traces to double to 14 million tuples and finally used the *data_len* column for a band join query. This query can be useful for the real-time monitoring of packet data information for a specific window.

Q 5: Query from Internet traces data set (band join)

```
SELECT *
FROM R.TRACES JOIN S.TRACES
    ON ABS(R.DATA_LEN - S.DATA_LEN) < 1000
WINDOW w AS (SLIDE INTERVAL 'D' ON 'W' )
```

- Stock market dataset contains prices for all tickers currently trading on NASDAQ [9]. Our band query tracks the observed high prices for the sliding window.

Q 6: Query from Stock market data set (self join)

```
SELECT *
FROM R.STOCK_MARKET
```

[8] https://data.mendeley.com/datasets/5pmnkshffm/2.
[9] https://www.kaggle.com/datasets/jacksoncrow/stock-market-dataset

```
    ON ABS (R1.HIGH-R2.HIGH) < Threshold
WINDOW w AS (SLIDE INTERVAL 'D' ON 'W' )
```

- 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 7.

Q 7: 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);
```

4.4. 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 9.

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

– **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 [54]. This is due to time latencies introduced by the network or straggling computational nodes. To measure that, we define the *completeness* as:

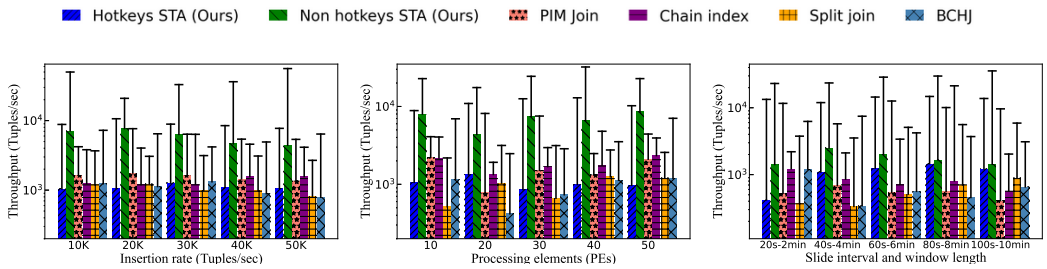
$$completeness = 1 - \frac{|X - \tilde{X}|}{|X|}$$

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 our inequality join algorithms with varying several parameters such as processing elements (PEs), insertion rate of tuples, sliding intervals, and window length. However, the standard size of these parameters while changing alternatives are 10 PEs, 10K tuples/sec, 10sec slide intervals, and 60sec window length. 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 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 [24] which has two layers for keeping track of hot-key elements. Roy et al. [24] state that throughput for the tuple processing algorithm is gradually increasing with increasing Zipf values in Section 7 [24]. 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.

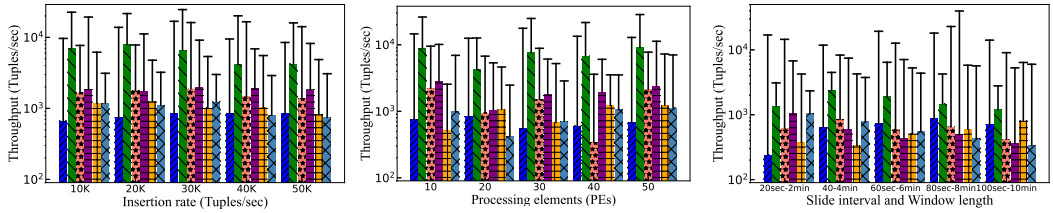


(a) Throughput with varying insertion rate.

(b) Throughput with the varying number of processing elements.

(c) Throughput with varying window length.

Figure 10: Throughput for tuple insertion into the sliding window for Q 3.



(a) Throughput with varying insertion rate.

(b) Throughput with varying number of processing elements.

(c) Throughput with varying window length.

Figure 11: Throughput for tuple lookup operation for Q 3.

5. Results and Discussion

In this section, we present and discuss the results of the experimental evaluation of STA-Join (STA), our novel stream-aware solution differentiating between hotkeys and non-hotkeys, against current state-of-the-art stream inequality join techniques, namely Partitioned In-Memory Merge-Tree (PIM), Chained Index (CI), Split Join (SJ), and Broadcast Hash Join (BCHJ). The different alternatives are tested on both real and synthetic data. In particular, we report the performance results measured on real data sets for self and cross join in Section 5.1, then for band join in Section 5.2. Section 5.3 describes instead the results achieved on synthetic data. Then, we focus on data partitioning and result completeness in Section 5.4. Finally, we briefly summarize the key findings of our experimental evaluation in Section 5.5.

5.1. Performance measurement on real data for self and cross join

Figures 10 and 11 depict the *average* and *maximum* throughput measured on the taxi dataset against the self-join query Q 3, focusing on tuple insertion

and lookup operations, respectively. Figure 10a reports the average and maximum throughput with varying insertion rates from 10K to 50K tuples per second, while other parameters are fixed to 10 processing elements (PEs) and a 2min sliding window. Figure 10a shows that the throughput with STA join for non-hotkeys is superior to all alternatives. Additionally, the maximum throughput for hotkeys is also higher than all existing schemes. Indeed, for the 10K insertion rate of tuples, throughput for non-hotkeys with STA-Join is 7x, 6.2x, 7.2x, and 7.3x better than PIM, CI, SJ, and BCHJ. Moreover, the maximum throughput of hotkeys is 2.09x, 2.31x, 2.39x, and 1.21x superior to alternative approaches. Similarly, this behavior is confirmed for 50K inserting tuples, with 3x, 2.8x, 5x, and 5.9x superior performance compared to the alternatives.

Figure 10b depicts the throughput by fixing the insertion rate to 10K tuples per second and varying the number of PEs from 10 to 50 with the slide interval of 20s and 2-minute window length. Results depict an analogous behavior, where non-hotkeys with STA-Join have 3x, 3.2x, 12x, and 6x superior performance compared to PIM, CI, SJ, and BCHJ for 10PEs. Moreover, hotkeys have 2.16x, 2.2x, 4.3x, and 1.27x superior maximum throughput than other join strategies. Similarly, for 50 PEs, it is 4.2x, 4.0x, 7.2x, and 7x for non-hotkeys, and for the maximum throughput of hotkeys, it is 2.3x, 4x, and 1.45x better than alternatives. Finally, Figure 10c depicts the throughput by fixing the insertion rate to 10K and processing elements to 10PEs by varying sliding interval and window length from 20sec to 100sec and 2-minutes to 10-minutes. It is depicted that non-hotkey throughput with STA-Join has 3.5x, 2.6x, 1.4x, and 2.2x superior performance compared to the PIM join, chain index, SJ, BCHJ for 100sec slide interval, and 10min sliding window size. Moreover, the average throughput for hotkeys with STA join is superior to alternatives.

Figure 11a depicts the average and maximum throughput for the tuple lookup operation with varying insertion rates. Results show that the throughput for non-hotkeys with STA Join achieves 4x, 3x, 5x, and 5.8x superior performance compared to PIM, CI, SJ, and BCHJ for an insertion rate of 50K tuples per second. Moreover, the maximum throughput of hotkeys with STA join is 1.03x against PIM join and it is 1.74x and 2.75x superior to split join and BCHJ respectively. Similarly, for 50 PEs, the performance of the proposed approach for non-frequent tuples is 4.5x, 4x, 6.3x, and 7x better than the alternatives, as depicted by Figure 11b. Additionally, the maximum throughput for the hotkeys is 1.6x, 1.2x, 7.7x, and 7.8x superior

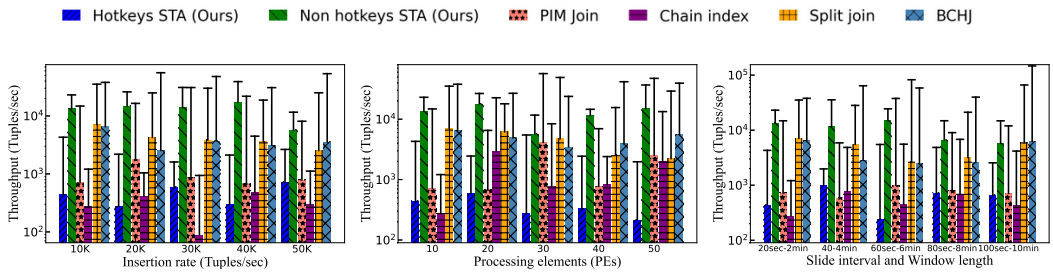
Table 4: Insertion (I) and Search (S) maximum processing latency (ms) for Q 3.

		Tuple insertion rate					Processing elements (PEs)					Window length				
		10K	20K	30K	40K	50K	10	20	30	40	50	2min	4min	6min	8min	10min
STA-H	I	34	37	30	63	30	34	10	39	28	14	9	16	20	33	20
	S	472	1648	1433	925	1695	344	904	809	423	441	224	207	331	383	309
STA-NH	I	98	48	149	237	57	61	26	31	74	62	152	28	88	49	107
	S	462	380	467	1260	2765	140	233	187	217	340	264	188	224	287	279
PIM-J	I	107	305	19	84	62	21	33	43	130	44	55	87	129	144	352
	S	1468	6579	4414	2286	2406	303	217	710	1918	8270	186	220	2965	2643	5316
CI	I	2063	12902	19702	6103	4041	321	177	30624	12284	1054	254	723	1215	15380	914
	S	1625	15401	19702	5370	5078	308	176	30632	12282	840	250	727	1215	15380	1118
SJ	I	3550	3903	5030	6403	3004	421	7794	4766	28762	11211	3202	3202	2185	3202	1202
	S	3550	3904	2162	6406	2508	415	7796	4773	28766	11212	7796	3203	2036	4000	1087
BCHJ	I	4268	4550	2593	2262	2338	585	7730	7427	23987	18777	20165	1393	4311	914	891
	S	4284	4580	4493	4342	3202	626	7736	7430	23988	18773	60162	4064	5685	5118	2710

to PIM, chain index, split join, and BCHJ algorithm. Figure 11c shows that STA-Join for non hotkeys performs 3x, 4x, 1.6x, and 4x better than the alternatives for a 10min sliding window size. Moreover, the maximum throughput for hotkeys with STA join is 1.5x, 2.67x, 2.2x, and 2.4x superior to alternative approaches.

Table 4 shows the maximum processing latency for inequality join algorithms for insertion and search operations for Q 3 with different input parameters. We measured hotkey (STA-H) and non-keys (STA-NH) latency for STA Join. The experimental results indicate that the hotkey latency for insertion is 3x better than PIM-join for a 10K insertion tuple rate, and 2x better for a 50K insertion rate. Similarly, for the search operation, we observe similar behavior in maximum processing latency, where the proposed approach is 3x and 1.5x superior to the PIM join strategy for 10K and 50K insertion rates of tuples. For other join algorithms such as chain index, SJ, and BCHJ, we observe that the variant of STA join such as STA-H and STA-NH, latency is far better than alternatives, as shown in Table 4. The maximum processing latency for hotkeys with STA join is 3x better for insertion and 18x better for search than PIM-Join for 50PEs. Additionally, the proposed approach that separate hotkeys and non-frequent keys outperforms alternative join strategies with varying processing elements. Table 4 also depicts the results with varying slide intervals from 20 seconds to 100 seconds and window lengths from 2 minutes to 10 minutes, showing that the proposed approach outperforms alternatives in all cases.

Figures 12 and 13 depict the maximum and average throughput of insertion and lookup operation on the BLOND dataset against the cross-join query Q 4. Figure 12a depicts that non-frequent keys with STA-join have 18x, 48x, 1.8x, and 2.06x superior performance than PIM, CI, SJ, and BCHJ for 10K insertion rate of tuples. Moreover, frequent keys with STA join show

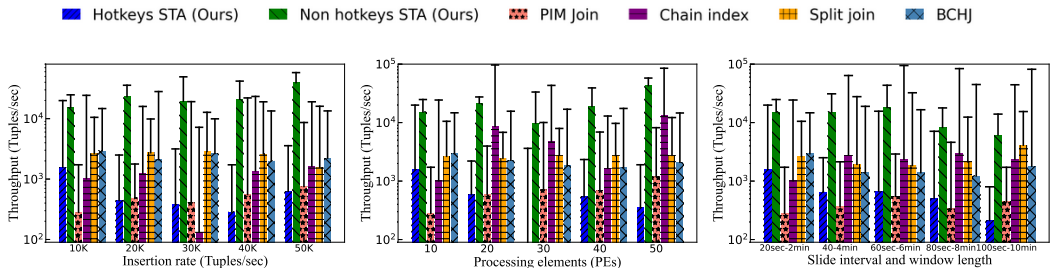


(a) Throughput with varying insertion rate.

(b) Throughput with the varying number of processing elements.

(c) Throughput with varying window length.

Figure 12: Throughput for tuple insertion into the sliding window for Q 4.



(a) Throughput with varying insertion rate.

(b) Throughput with the varying number of processing elements.

(c) Throughput with varying window length and slide interval.

Figure 13: Throughput for tuple lookup operation for Q 4.

analogous performance behavior to alternatives. Similarly, non-frequent keys with STA are 7.2x, 19x, 2.2x, and 1.6x superior to alternatives for the 50K insertion rate of tuples. However, a small performance degradation is observed for frequent key than alternative strategies. Figure 12b illustrates insertion throughput with varying processing elements. Results show that the proposed approach has 26x, 6x, 2.7x, and 3.6x time superior performance than PIM, CI, SJ, and BCHJ for 20 PEs. Additionally, it is 5.9x, 7.5x, 6.7x, and 2.6x time better for 50 PEs. Analogous performance improvement for non-frequent hotkeys with STA is noted for changing the sliding window, where the proposed approach has 14x, 28x, 5.6x, and 5.9x improved than the alternative for 40sec slide interval and 4min of window size. Similarly, it has 8.1x, 13x, 1.5x, and 1.6x better performance than the alternative for the 10-minute sliding window as depicted by Figure 12c.

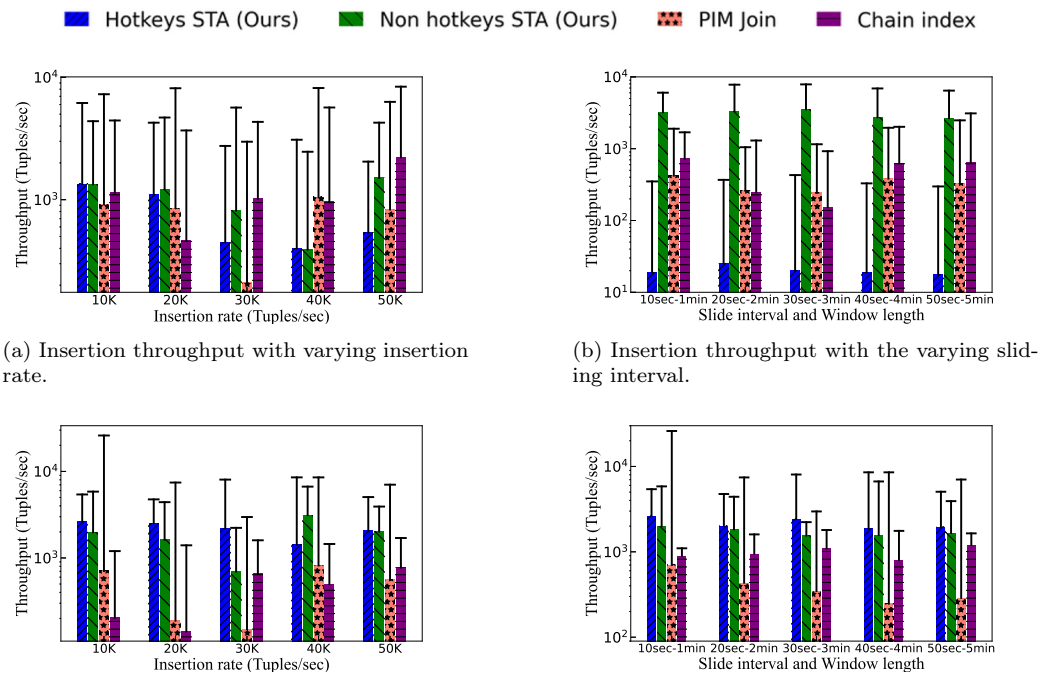
Figure 13a depicts the lookup operation with varying insertion rates from 10K to 50K processing elements. Results depict that the overall throughput

Table 5: Insertion (I) and Search (S) maximum processing latency (ms) for Q 4.

		Tuple insertion rate					Processing elements					Window length				
		10K	20K	30K	40K	50K	10	20	30	40	50	2min	4min	6min	8min	10min
STA-H	I	41	30	10	10	45	41	26	47	30	19	41	28	16	41	12
	S	321	260	331	222	197	321	169	139	514	362	321	131	257	289	217
STA-NH	I	78	479	614	73	271	78	100	126	99	58	78	336	236	136	87
	S	50	74	64	21	105	50	87	12	57	36	50	21	86	26	35
PIM-J	I	218	313	197	166	168	218	293	573	4272	6602	218	152	576	110	157
	S	1331	864	645	278	1198	1331	2549	1477	6299	2302	1331	2467	1068	1373	1075
CI	I	663	826	684	653	1920	663	3285	1906	1428	2660	663	1376	518	2443	3126
	S	663	826	1302	342	624	663	32833	1938	1692	2660	663	1394	524	2441	3226
SJ	I	80	98	83	285	47	80	142	119	82	321	80	68	34	48	39
	S	143	124	379	149	275	143	339	632	9610	1480	143	352	130	118	160
BCHJ	I	45	48	1800	1315	60	45	50	1311	448	1689	45	57	41	3291	26
	S	173	211	196	149	148	173	1029	1028	1444	1958	173	236	245	246	374

of STA join is 5.5x, 15x, 5.74x, and 5.17x better than the alternatives for a 10K insertion rate, moreover, it is 6.2x, 24x, 25x, and 17x superior for the 50K insertion rate. Similarly, Figure 13b depicts the throughput of the lookup operation with increasing PEs. Results show that the overall throughput of STA join (STA-H and STA-NH collective) has 4.2x, 2.4, 8.6x, and 9.4x for 20 PEs and 3.5x, 3.2x, 15.5x, and 20.4x for 50 PEs superior performance than alternatives. Moreover, Figure 13c depicts the throughput of the lookup operation with increasing window length. Results show the superior performance of proposed STA join to 2x, 5.4x, 7.8x, and 10.8x for 40sec and 4min window and it is 13x, 2.5x, 1.4x, and 3.43x for 10min sliding window.

Table 5 displays the maximum processing latency for tuple insertion and lookup operations for cross-join query Q 4. The results of the table illustrate the maximum processing latency for hotkeys or non-hotkeys is better than other alternative join algorithms. The table results show that the hotkeys with STA-Join have 5.3x and 6.02x and non-frequent keys have 2.7x and 2.2x lower insertion costs compared to PIM join for input rates of 10K and 50K tuples. Similarly, frequent keys with STA are 3.8x and 34x better than PIM join for 10 and 50 processing elements (PEs). Furthermore, the maximum latency for the STA-hot key is 5x and 12x superior to the PIM join strategy for sliding windows of 2-minutes and 10-minutes. Similar performance improvements are observed for alternative stream inequality join algorithms compared to the proposed approach. Table 5 also shows that the maximum latency for lookup operation for the hotkeys of STA-Join is 4.14x and 6.06x lower than PIM-join for 10K and 50K input inserting tuples. Additionally, it is 15x and 6.35x for 20 and 50 PEs, and 18x and 4.9x for sliding windows of 40 seconds and 4 minutes, and 100 seconds and 10 minutes. Similarly, Table 5 indicates that the proposed STA-join algorithm outperforms other alternative algorithms. Analogous performance improvement is noted for



(a) Insertion throughput with varying insertion rate.

(b) Insertion throughput with the varying sliding interval.

(c) Lookup throughput with varying insertion rate.

(d) Lookup throughput with varying window length.

Figure 14: Throughput for tuple insertion and lookup into the sliding window for band join Q 5.

non-frequent keys with STA for most of the cases.

5.2. Performance measurement on real data for band join

Figure 14 depicts the maximum and average throughput for the insert and lookup operations on the internet traces dataset against the band-join query Q 5. Figure 14a shows the insertion cost of increasing the input insertion rate of tuples. It is depicted that the average throughput for STA-hotkeys and non-hotkeys is 1300 tuples/sec and 1336 tuples/sec, respectively. Similarly, it is observed that STA-join achieves 1.4x and 3.1x better performance than PIM and CI join strategies for a 10K input insertion rate, and 4x and 1.2x better performance for a 20K input insertion rate. Figure 14b depicts the throughput of insertion with varying slide intervals and sliding windows. The average throughput for STA-hotkeys is 1000 tuples/sec, while it is 1120 tuples/sec for non-hotkey tuples. Additionally, the proposed algorithm shows 1.2x and 1.4x better performance than PIM and CI for a 1-minute sliding

Table 6: Insertion (I) and Search (S) maximum processing latency (ms) for Q 5.

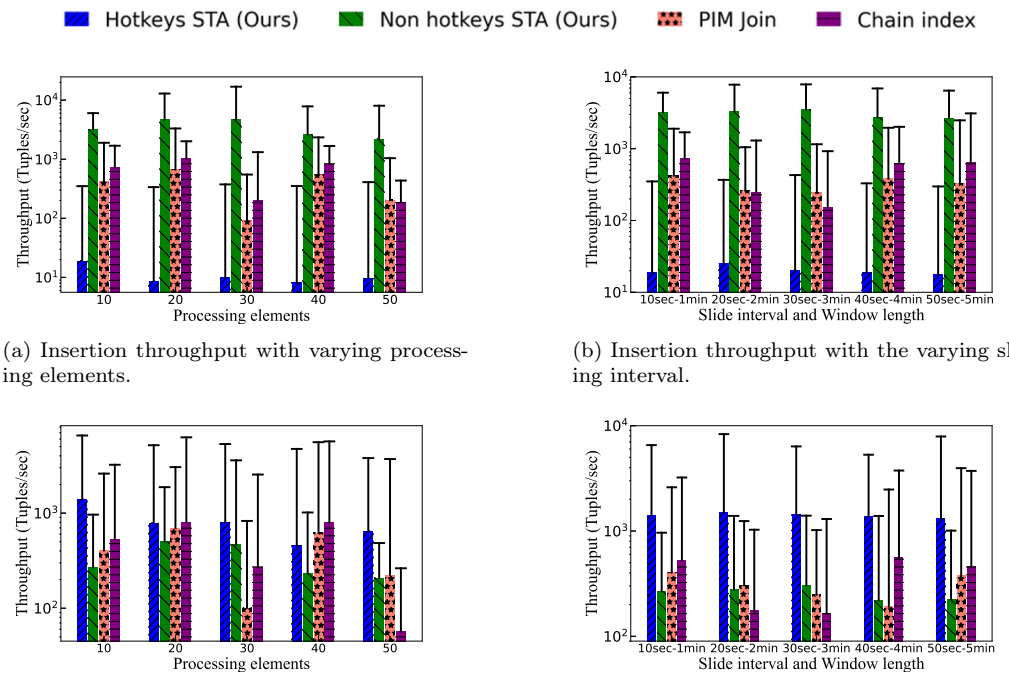
		Tuple insertion rate					Window length				
		10K	**20K**	**30K**	**40K**	**50K**	**2min**	**4min**	**6min**	**8min**	**10min**
STA-H	**I**	3	31	2	3	30	3	28	30	5	44
	S	23	53	23	80	49	23	41	412	53	398
STA-NH	**I**	19	21	28	12	23	19	139	100	19	27
	S	48	56	20	23	32	48	79	136	69	124
PIM-J	**I**	108	172	210	193	222	108	256	359	433	470
	S	2866	10307	941	11675	21383	2866	13134	15387	14836	12019
CI	**I**	407	566	700	916	615	404	356	410	409	550
	S	40711	56441	26647	19154	31628	40711	37932	42476	40409	560

window, and 1.3x and 1.8x better performance for a 5-minute sliding window.

Figure 14c depicts the lookup cost for increasing insertion rates of tuples. The average throughput for STA-hotkeys is observed to be 2000 tuples/sec, while for non-hotkeys it is 1500 tuples/sec. The proposed STA-Join achieves 3x and 9x better performance than PIM and CI for a 10K insertion rate, and 4x and 3x better performance for a 50K insertion rate. Figure 14d shows the lookup throughput with varying sliding windows. The average throughput for STA-hot keys is 1800 tuples/sec, and for non-hotkeys, it is 1500 tuples/sec. Additionally, STA-Join demonstrates 4x and 1.8x better performance than PIM and chain index for a 2-minute sliding window, and 7.3x and 1.2x better performance for a 5-minute sliding window.

Table 6 presents the maximum processing latency for stream inequality strategies for query Q 5. For the insert operation of STA-Join, the maximum latency is measured at 3ms for a 10K insertion rate, whereas for PIM and CI, it is significantly higher at 108ms and 407ms, respectively. Similarly, for a 50K insertion rate, STA-Join exhibits a latency of 193ms, whereas PIM-Join and chain index show latency of 916ms and 193ms, respectively. Analogous performance superiority is observed for the lookup operation, as demonstrated in Table 6. When changing the sliding window, the proposed join strategy exhibits latency that is 10x and 14x lower than that of PIM and chain index for insertion operations. Moreover, for lookup operations, it shows latency that is 30x and 1.3x lower than the alternatives.

Figure 15 depicts the insertion and lookup throughput for the band-join query Q 6. Figure 15a depicts that the insertion throughput for non-hot keys with STA-join outperforms PIM and chain index solutions by 7.6x and 4.4x for 10PEs. Additionally, the maximum throughput of hotkeys with STA-Join approaches the average throughput of PIM and chained index. Similarly, for 50PEs, the non-hot keys throughput with STA-Join is 10x and 11.6x better than alternatives. Figure 15b depicts the insertion throughput with increas-



(a) Insertion throughput with varying processing elements.

(b) Insertion throughput with the varying sliding interval.

(c) Lookup throughput with varying processing elements.

(d) Lookup throughput with varying window length.

Figure 15: Throughput for tuple insertion and lookup into the sliding window for band join Q 6.

ing sliding windows, showing that non-hotkeys with STA join outperform alternatives by 12.8x and 13.4x for a 2-minute sliding window. Similarly, the maximum throughput of hotkeys approaches the average throughput of PIM and chain indexes. The proposed approach has 8.16x and 4.17x superior performance to PIM and chain indexes.

Figure 15c shows the lookup throughput as the number of PEs increases. The results indicate that hotkeys perform 3.4x and 2.6x better than the alternatives for 10 PEs. Furthermore, the average throughput of non-keys is approaching the average throughput of PIM and chain indexes. Similar performance is observed for 50 PEs, where the proposed approach outperforms the alternatives by 2.9x and 11.4x. Additionally, Figure 15d demonstrates that hotkeys with STA have 4.8x and 3.5x better performance for 2-minute sliding windows, and 3.4x and 2.8x better for 5-minute sliding windows. However, the performance of non-hotkeys is similar to alternative stream join strategies.

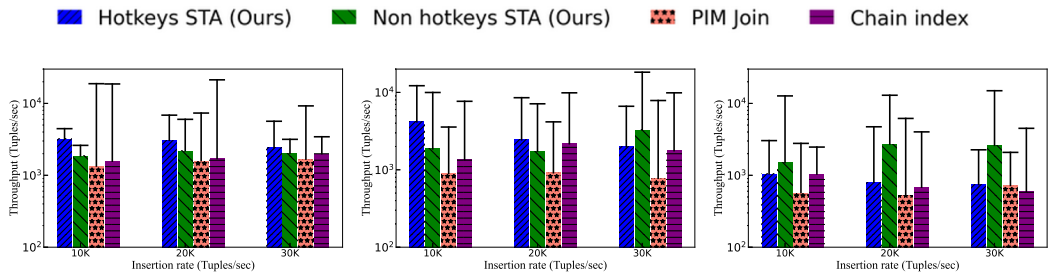
Table 7 illustrates the comparative processing latencies between the proposed STA-Join method and alternative techniques such as PIM and chained index (CI) for query Q 6. The results reveal that the proposed method exhibits significantly reduced maximum latencies, particularly evident in scenarios involving 10 processing elements (PEs), with insertion latency at only 4ms and search latency at 45ms. For non-hot-keys, the latency is similarly impressive at 93ms for insertion and 206sec for search. In comparison, PIM demonstrates latencies of 77ms and 12sec, while chained index records 8.7sec and 8.1sec for insertion and search, respectively. Moreover, Table 7 highlights consistent performance enhancements across various scenarios. Specifically, for a sliding interval of 50sec and a 5-minute window, the proposed method maintains low latencies, with insertion and lookup latencies for hotkeys at 4ms and 106ms, respectively, and 3sec and 200sec for non-hotkeys. In contrast, PIM and chained index exhibit higher latencies, with insertion and search operations showing values of 22ms and 15sec for PIM, and 8sec and 15.5sec for chained index, respectively. Overall, the findings presented in Table 7 underscore the superior performance of the proposed approach across various sliding window configurations, validating its efficacy in real-world application scenarios.

5.3. Performance measurement on synthetic Zipf data

Figure 16 illustrates the average and maximum throughput with varying *Zipf* data from Z=1.0 to Z=2.0 for query Q 7. In Figure 16a, the average throughput for Z=1.0 is depicted. For hotkeys with STA join, there is a notable improvement, showing 2.3x and 2.05x better performance compared to PIM and chain index, respectively, at a 10K input insertion rate. Similarly, the improvement is 1.3x and 1.7x for non-hot keys. Moving to a 20K input insertion rate, the performance gains persist, with hotkeys utilizing STA join achieving 1.9x and 1.7x better throughput compared to alternative approaches, and non-hot keys showing 1.3x and 1.2x improvement. Notably,

Table 7: Insertion (I) and Search (S) maximum processing latency (ms) for Q 6.

		Processing elements					**Window length**				
		10	**20**	**30**	**40**	**50**	**2min**	**4min**	**6min**	**8min**	**10min**
STA-H	**I**	4	32	8	1	38	4	5	12	3	4
	S	45	53	81	4852	193	45	59	66	36	106
STA-NH	**I**	93	49	52	32	56	93	16	7	33	3
	S	206	295	1682	2850	4114	206	76	124	98	200
PIM-J	**I**	77	19	17	34	31	77	6	59	39	22
	S	12287	2675	56665	19500	66561	12287	5074	3817	3792	8694
CI	**I**	8704	2291	16854	19666	3622	8704	2416	2482	10151	15493
	S	8721	2291	16854	19667	13616	8721	2416	2482	9531	15510

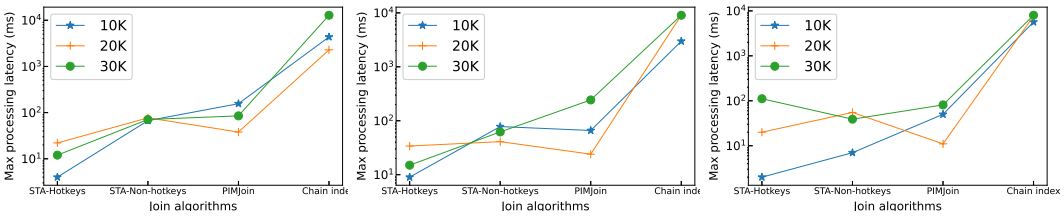


(a) Throughput with varying insertion rate for Zipf 1.0

(b) Throughput with the varying insertion rate for Zipf 1.5

(c) Throughput with the varying insertion rate for Zipf 2.0

Figure 16: Throughput for Zipf data.



(a) Insertion latency with varying insertion rate for Zipf 1.0

(b) Insertion latency with varying insertion rate for Zipf 1.5

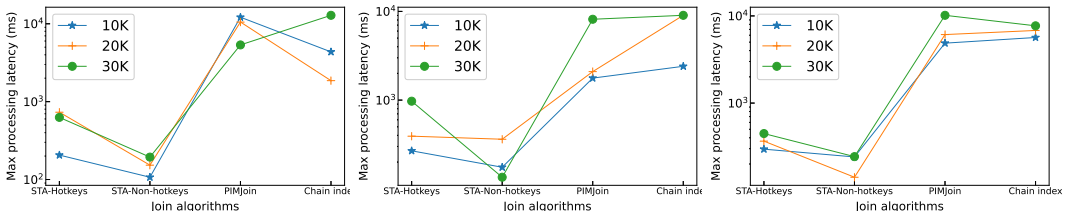
(c) Insertion latency with varying insertion rate for Zipf 2.0

Figure 17: Insertion latency for Zipf data.

the throughput is 1.45x and 1.22x higher for hotkeys and 1.21x and 1.09x for non-hotkeys, outperforming PIM and chained index methods, respectively.

Figure 16b shows the average and maximum throughput for synthetic data with z=1.5. Results depict that hotkeys with STA join have 1.8x and 1.03x for 10K, 1.5x and 1.2x for 20k, and it is 1.3x and 1.35x for 30k better than PIM and chain indexes. Similarly, for the non-hotkeys, it is 3x and 1.5x for 10K, 5.3x and 4.2x for 20K, and 3.4x and 4.2x for 30k superior to PIM and chain indexes. Moreover, Figure 16c shows that hotkeys with STA join have 4.5x and 3.8x for 10K, 2.6x and 1.4x for 20K, and 2.9x and 1.5x superior to alternative join strategies. Similarly, analogous improved join performance is observed for non-hotkeys.

Figure 17 illustrates the insertion latency across varying Zipf distributions from Z=1.0 to Z=2.0. In this experiment, the tuple input rate ranges from 10K to 30K tuples per second, while maintaining 10 processing elements (PEs) and a fixed window size of 10 seconds with a 1-minute window length. In Figure 17a, it is observed that the maximum insertion latency for hotkeys



(a) Lookup latency with varying insertion rate for Zipf 1.0

(b) Lookup latency with varying insertion rate for Zipf 1.5

(c) Lookup latency with varying insertion rate for Zipf 2.0

Figure 18: Lookup latency for Zipf data.

using the STA algorithm is 4ms, whereas, for non-hotkeys, it is 67ms for Z=1.0. In contrast, PIM and CI exhibit latencies of 156ms and 4.3s, respectively, at a 10K insertion rate. The analogous superior performance of the STA join algorithm is noted at higher insertion rates. Moving to Figure 17b, it is evident that the proposed approach for hotkeys achieves a maximum latency of 15ms, while for non-hotkeys, it is 62ms for a 30K insertion rate of tuples with Z=1.5. Comparatively, the maximum latencies for PIM join and chain indexes are 243ms and 9.4s, respectively. Furthermore, in Figure 17c, the STA join algorithm continues to outperform alternatives for Z=2.0.

Figure 18 illustrates the lookup latency across varying Zipf distributions from Z=1.0 to Z=2.0, considering different insertion rates. In Figure 18a, it's evident that for Z=1.0, the maximum latency for hotkeys employing STA approaches 628ms, while for non-hotkeys, it stands at 194ms. In comparison, PIM and chain index approaches exhibit latencies of 5.3s and 12.4s, respectively, with a 30K insertion rate. Moving to Figure 18b, the maximum latency with hotkeys is 972ms, whereas for non-hotkeys, it's 135ms at Z=1.5. Conversely, PIM and chain index methods demonstrate latencies of 8s and 9s, respectively, with a 30K insertion rate. Furthermore, Figure 18c illustrates that the proposed STA join achieves latencies of 447ms and 243ms for its variant at Z=2.0. In contrast, PIM and chain indexes record latencies of 10s and 7s, respectively, with a 30K insertion rate. Notably, it's observed that the maximum latency incurred by the STA join variant is superior to alternatives across all scenarios.

The stream-aware solution is performing better due to several reasons. Our solution consists of both data partitioning and stream indexing structure for a complete stream join process. BCHJ replicates the window to all processing elements which is less memory efficient. Split join requires search-

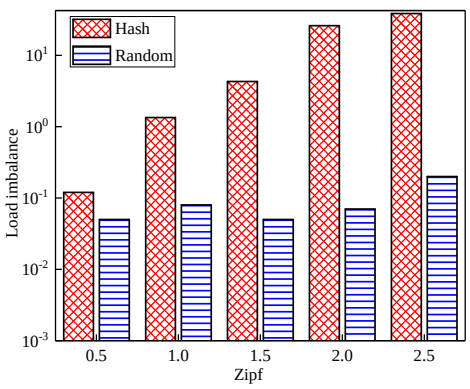


Figure 19: Load imbalance comparison for hash and random data partitioning (the lower the better).

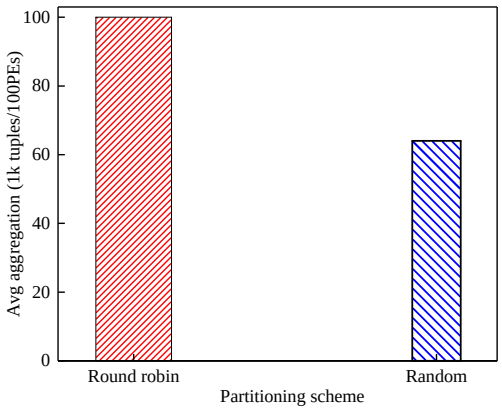


Figure 20: Aggregation of tuples for round-robin and random partitioning schemes (the lower the better).

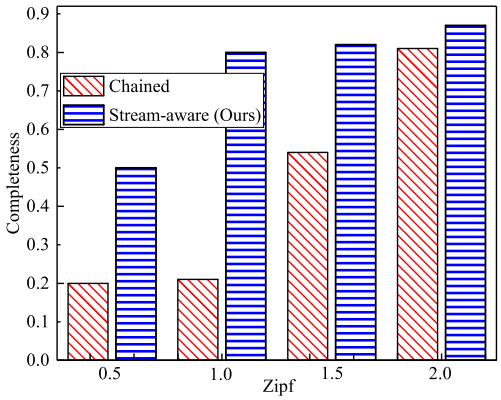


Figure 21: Completeness with 10 PEs and 1sec checkpoint (the higher the better).

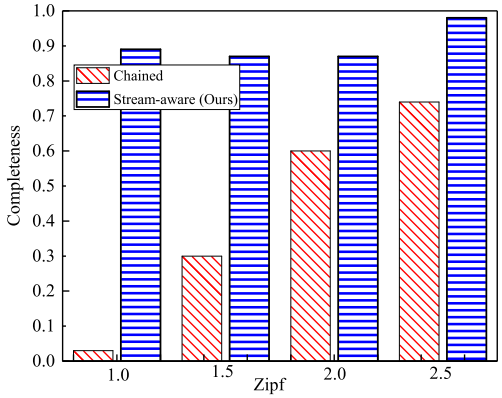


Figure 22: Completeness with 100 PEs and 5sec check pointing (the higher the better).

ing 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. Detail description on results discussion can be found in Section 5.5

5.4. 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 19 shows the comparison of load imbalance among PEs for hash-based and random partitioning strategies. The result depicts that with an increase in Zipf 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 20 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.

Figures 21 and 22 show the result for completeness of stream-aware solution with chained index scheme with varying Zipf data. Figure 21 depicts the completeness results with 10 PEs with a checkpoint of 1 second. 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 22 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 updates on changes in the frequencies of keys inside this structure. Only non-hot keys data structure share their state among PEs, reducing the chance of the incomplete result set for any on-the-fly search query.

5.5. Key findings

The results of experiments conducted on both real-world and synthetic data sets show that inequality joins with STA-Join produce better outcomes compared to cache-sensitive search tree-based distributed inequality join,

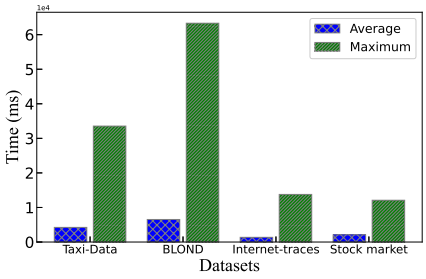


Figure 23: PIM-Join merging overhead for real-time datasets

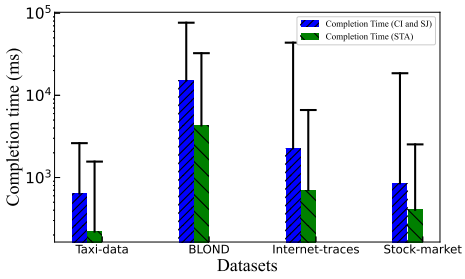


Figure 24: Completion time with real-time datasets

chain indexes, partitioner-based split join, and broadcast hash-join. STA-Join operates by identifying key frequencies in real-time and treating them differently based on their popularity, which sets it apart from other methods. Hotkeys are distributed using a power-of-two-choice data partitioning strategy to two dedicated downstream processing elements, effectively distributing the load of the hotkeys payload. Non-hotkeys, on the other hand, are partitioned using random data partitioning, which reduces the overhead of hash computation for each key. Additionally, the experimental results indicate that random partitioning is more effective for load balancing than hash partitioning, and its performance continues to improve as the *Zipf* increases. We consider both insert and search throughput and latencies for the performance measurement of the proposed approach.

Distributed PIM-join includes a two-level data structure such as an immutable CSS-search tree and a linked set of mutable index structures (B^+tree). A new tuple first explores the CSS-tree data structures, identifies its corresponding results, then starts search from the sub-index pointed by the CSS-tree level. Search operation using CSS-tree is very expensive than traditional B^+tree structure due to the implicit addressing, However, indexing is more expensive for the real-time operation. Moreover, the PIM-tree design includes a merge operation, where tuples from the linked set of mutable data structures are merged into an immutable CSS-tree. In distributed processing, where many distributed processing instances hold the content of sliding windows, the extra overhead of merging increases.

The merging overhead of tuples for various datasets in distributed PIM-join is illustrated in Figure 23 for benchmark experimental setup. The results show the average merging time, which encompasses the entire duration from

the start of the merging process until the tuples are merged into the CSS-tree. Additionally, it includes the evaluation of tuples during the merge operation. The average time it takes to merge tuples ranges from 1.3 seconds to 6.5 seconds. However, there is a significant variation in the maximum time across datasets due to different levels of selectivity and query complexity. Additionally, scaling out the processing of the mutable indexing data structures is challenging because all of these index structures are organized and linked in the form of ranges. STA-Join is more scalable as non-hotkeys are held by many distributed processing elements. Further, it does not present an overhead for merging during join processing. STA-Join presents instead a little overhead for state-synchronizing its chain-index structure for non-hotkeys among processing elements and a little degradation of performance for hotkeys to keep track of the popularity of real-time tuples. However, STA-Join assumes that most of the dataset follows a skewed distribution, where the proposed approach has a dedicated index structure on distributed PEs. Thus, the overall cost of index update is reduced if compared to the PIM-Join structure, which needs to explore two structures for each key insertion.

In chain-index and split-join, a new tuple is only inserted into the sub-index data structure that exists on single processing elements (PEs). This procedure reduces the index update rate compared to complete indexing for large-size sliding windows. However, a search is performed on all sub-indexes, whether they are chained (chain index) or exist independently on different processing elements. In this study, we follow the semantics of distributed stream processing frameworks, which consist of the number of downstream processing elements. Chain-index approach builds a separate chain index on each PE and updates it locally. Moreover, we propose a state synchronization mechanism for the chain index to update the window state among distributed processing elements of the inequality operator. A new tuple is inserted on any downstream processing element dedicated to streaming window content; however, the lookup operation requires searching all downstream processing instances for completeness, both for chain-index and split-join.

Figure 24 depicts the difference between STA and these sub-index-based strategies. Results show that the proposed approach has an average of 2.9x, 3.5x, 3.2x, and 2.09x better completion time for taxi data, BLOND, Internet traces, and stock market datasets in the benchmark experimental setup. Moreover, it is depicted that the maximum time taken during this setup is also better than the alternative. The reason is that sliding window contents are held in many distributed processing elements (10 PEs in the benchmark

setting). A new tuple needs to expedite all of these data structures. In split-join and PIM-join, the result completion time depends on the results of all upstream processing elements of the inequality operator. However, in the proposed join strategy, processing elements dedicated to non-hotkeys continually synchronize their state of data structure periodically, so a consistent state of the index structure exists on all processing elements. This process reduces the overhead of waiting for results from all upstream processing elements and helps to perform better in throughput and latency.

The broadcast hash join (BCHJ) involves replicating the entire stream to all downstream processing instances of the inequality operator. The sliding window contents are stored in a list, with new tuples always being inserted at the end of the index data structure. This process is the same for the SJ. One of the main advantages of this procedure is that the cost of inserting a tuple is negligible (O(1)) on each processing element. However, the search time increases linearly with the size of the sliding window. Additionally, removing tuples from the sliding window is a time-consuming procedure, as it involves removing expired tuples from the start of the window at the slide interval. Moreover, highly skewed data creates extra overhead, as more tuples are performing inequality join operations on the same processing element, increasing the wait time for new tuples for completion of results. On the other hand, for the proposed STA-Join, the sliding window contents are stored using an indexing data structure, which improves the search procedures for new tuples. Furthermore, tuples are removed in a coarse granular way, where the entire expired index structure is removed from memory in constant time (O(n)).

6. 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 to 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 mitigate this issue, but implementing an efficient dynamic repartitioning technique remains challenging. 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.

Declaration

During the preparation of this work, the author(s) used Premium Grammarly and GPT to improve sentence structure or grammar mistakes. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Acknowledgement

We gratefully acknowledge the support provided by Rif.PA 2023- 20467/RER with grant no CUP E83C23002540002, FAR_DIP_2023_ DIEF-SIMONINI

with grant no CUP E93C23000280005, and "Discount Quality for Responsible Data Science: Human-in-the-Loop for Quality Data" with project grant no CUP 202248FWFS.

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] R. Li, W. Gatterbauer, M. Riedewald, Near-optimal distributed band-joins through recursive partitioning, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 2375–2390.

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] M. Li, H. Wang, H. Dai, M. Li, R. Gu, F. Chen, Z. Chen, S. Li, Q. Liu, G. Chen, A survey of multi-dimensional indexes: Past and future trends, IEEE Transactions on Knowledge and Data Engineering (2024).

[38] S. A. Shaikh, K. Mariam, H. Kitagawa, K.-S. Kim, Geoflink: A distributed and scalable framework for the real-time processing of spatial streams, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 3149–3156.

[39] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, 1984, pp. 47–57.

[40] D. Zhang, Z. Chang, D. Yang, D. Li, K.-L. Tan, K. Chen, G. Chen, Squid: subtrajectory query in trillion-scale gps database, The VLDB Journal 32 (4) (2023) 887–904.

[41] Z. Fang, S. Gong, L. Chen, J. Xu, Y. Gao, C. S. Jensen, Ghost: A general framework for high-performance online similarity queries over distributed trajectory streams, Proceedings of the ACM on Management of Data 1 (2) (2023) 1–25.

[42] F. Zhang, H. Chen, H. Jin, Simois: A scalable distributed stream join system with skewed workloads, in: 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019, IEEE, 2019, pp. 176–185.

[43] 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.

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.