Parallel Index-based Stream Join on a Multicore CPU

Amirhesam Shahvarani, Hans-Arno Jacobsen
Technische Universität München
Munich, Germany
ah.shahvarani@tum.de

ABSTRACT

Indexing sliding window content to enhance the performance of streaming queries can be greatly improved by utilizing the computational capabilities of a multicore processor. Conventional indexing data structures optimized for frequent search queries on a prestored dataset do not meet the demands of indexing highly dynamic data as in streaming environments. In this paper, we introduce an index data structure, called the *partitioned in-memory merge tree*, to address the challenges that arise when indexing highly dynamic data, which are common in streaming settings. Utilizing the specific pattern of streaming data and the distribution of queries, we propose a low-cost and effective concurrency control mechanism to meet the demands of high-rate update queries. To complement the index, we design an algorithm to realize a parallel index-based stream join that exploits the computational power of multicore processors. Our experiments using an octa-core processor show that our parallel stream join achieves up to 5.5 times higher throughput than a single-threaded approach.

CCS CONCEPTS

• **Information systems → Data structures**; • **Computing methodologies → Parallel algorithms**.

ACM Reference Format:
Amirhesam Shahvarani, Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380576

1 INTRODUCTION

For a growing class of data management applications, such as algorithmic trading [27], fraud detection [47], social network analysis [11], and real-time data analytics [40], an information source is available as a transient, in-memory, real-time, and continuous sequence of tuples (also known as a *data stream*) rather than as a persistently disk-stored dataset [9]. In these applications, processing is mostly performed using long-running queries known as *continuous queries* [2]. Although its size is steadily increasing, the limited capacity of system memory is a general obstacle to processing potentially infinite data streams. To address this problem, the scope of continuous queries is typically limited to a *sliding window* that limits the number of tuples to process at any one point in time. The window is either defined over a fixed number of tuples (*count based*) or is a function of time (*time based*).

Indexing the content of the sliding window is necessary to eliminate memory-intensive scans during searches and to enhance the performance of window queries, as in conventional databases [15]. In terms of indexing data structures, hash tables are generally faster than tree-based data structures for both update and search operations. However, hash-based indexes are applicable only for operations that use equality predicates since the logical order of indexed values is not preserved by a hash table. Consequently, tree-based indexing is essential for applications that analyze continuous variables and employ nonequality predicates [46]. Thus, in this paper, we focus on tree-based indexing approaches, which are also applicable to operators that use nonequality predicates.

Due to the distinct characteristics of the data flow in streaming settings, the indexing data structures designed for conventional databases, such as B^+-Tree, are not efficient for indexing streaming data. Data in streaming settings are highly dynamic, and the underlying indexes must be continuously updated. In contrast to indexing in conventional databases, where search is among the most frequent and critical operations, support for an efficient index update is vital in a streaming setting. Moreover, tuple movement in sliding windows follows a specific pattern of arrival and departure that could be utilized to improve indexing performance.

In addition to the index maintenance overhead arising from data dynamics, proposing a concurrency control (CC) scheme for multithreaded indexing that handles frequent updates is also a challenging endeavor. In conventional databases, the index update rate is lower than the index lookup rate, and CC schemes are designed accordingly. Therefore, these approaches are suboptimal for indexing highly dynamic data, such as sliding windows, for which they have not been designed. Thus, dedicated solutions are desired to coordinate dynamic workloads with highly concurrent index updates. These issues are further exacerbated because the continued leveraging of the computational power of multicore processors is becoming inevitable in high-performance stream processing. The shift in processor design from the single-core to the multicore paradigm has initiated widespread efforts to leverage parallelism in all types of applications to enhance performance, and stream processing is no exception [14, 36, 38, 39].

In terms of the underlying hardware, stream processing systems (SPSs) are divided into two categories, single-node and multinode. Single-node SPSs are designed to exploit the computation power of a single high-performance machine and are optimized for *scale-up* execution, such as Trill [8], StreamBox [26] and Saber [18]. In contrast, multinode SPSs are intended to exploit a multinode cluster. A group of multinode SPSs, such as Storm [42], Spark [44] and Flink [6], are optimized for *scale-out* execution and rely on massive parallelism in the workload and the producer-consumer pattern to distribute tasks among nodes. As a consequence, these systems achieve suboptimal single-node performance in comparison with a single-node SPS or multinode SPSs optimized for both scale-up and scale-out execution, such as IBM System S [12, 16]. With advances in modern single-node servers, scale-up optimized solutions become an interesting alternative for high-throughput and low-latency stream processing for many applications [45].

Thus, in this paper, we address the challenges of parallel tree-based sliding window indexing, which is designed to exploit a multicore processor on the basis of uniform memory access. The distinct characteristics of streaming data motivated us to reconsider how to parallelize a stream index and design a novel mechanism dedicated to a streaming setting. We propose a two-stage data structure based on two known techniques, data partitioning and delta updating, called the *partitioned in-memory merge tree* (PIM-Tree), that consists of a mutable component and an immutable component to address the challenges inherent to concurrent indexing in highly dynamic settings. The mutable component in PIM-Tree is partitioned into multiple disjoint ranges that can dynamically adapt to the range of the streaming tuple values. This multipartition design enables PIM-Tree to benefit from the distribution of queries to reduce potential conflicts among queries and to support parallel index lookup and update through a simple and low-cost CC method. Moreover, leveraging a coarse-grained tuple disposal scheme based on this two-stage design, PIM-Tree significantly reduces the amortized cost of sliding window updates relative to individual tuple updates in conventional indexes such as a B$^+$-Tree.

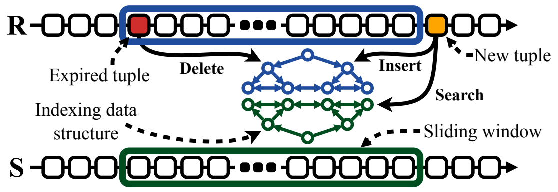

Figure 1: Index-based window join.

By combining these two techniques PIM-Tree outperforms state-of-the-art indexing approaches in both single- and multithreaded settings.

To validate our indexing approach, we evaluate it in the context of performing a window band join. Stream join is a fundamental operation for performing real-time analytics by correlating the tuples of two streams, and it is among the most computationally intensive tasks in SPSs. Nonetheless, our indexing approach is generic and applies equally well to other streaming operations.

To complement our data structure, we develop a parallel window band join algorithm based on dynamic load balancing and shared sliding window indexes. These features enable our join algorithm to perform a parallel window join using an arbitrary number of available threads. Thus, the number of threads assigned for a join operation can be adjusted at run time based on the workload and the hardware available. Moreover, our join algorithm preserves the order of the result tuples such that if tuple t_1 arrives before t_2, the join result of tuple t_1 will be propagated into the output stream before that for t_2.

The evaluation results indicate that our multithreaded join algorithm using PIM-Tree achieves up to 5.6 times higher throughput than our single-threaded implementation using an octa-core processor. Moreover, a single-threaded stream band join using PIM-Tree is 60% faster on average than that using B$^+$-Tree, which demonstrates the efficiency of our data structure for stream indexing applications. Compared with a stream band join using the state-of-the-art parallel indexing tree index Bw-Tree [23], using PIM-Tree improves the system performance by a factor of 2.6 on average.

In summary, the contributions of this paper are fourfold: (1) We propose PIM-Tree, a novel two-stage data structure designed to address the challenges of indexing highly dynamic data, which outperforms state-of-the-art indexing methods in the application of window joins in both single- and multithreaded settings. (2) We develop an analytical model to compare the costs of window joins using the indexing approaches studied in this paper to provide better insight into our design decisions. (3) We propose a parallel index-based window join (IBWJ) algorithm that addresses the challenges arising from using a shared index in a concurrent manner. (4) We conduct an extensive experimental study of IBWJ employing PIM-Tree and provide a detailed quantitative comparison with state-of-the-art approaches.

2 INDEX-BASED WINDOW JOIN

In this section, we define the stream join operator semantics and study IBWJ using three existing indexing approaches, including B$^+$-Tree, chain-index and round-robin partitioning, to highlight the challenges of sliding window indexing and the shortcomings of existing methods. We also provide an analytical comparison of processing a tuple using each approach to provide better insight into each mechanism and highlight their differences from our approach. The notation that we use in this paper is as follows.

w	:	Size of sliding window.
τ_c	:	Time complexity of comparing two tuples.
σ	:	Join selectivity ($0 \leq \sigma \leq 1$).
σ_s	:	Match rate ($w \times \sigma$).
f_T	:	Inner node fan-out of a tree of type 'T'.
λ_T^O	:	Time complexity of performing an operation (O: Insert, Search, Delete) on a node of a tree of type 'T'.

Throughout the remainder of this paper, λ_b^s, λ_b^i and λ_b^d denote the time complexities of search, insert and delete operations at each node of B$^+$-Tree, respectively, and f_b denotes the inner node fan-out of B$^+$-Tree.

2.1 Window Join

The common types of sliding windows are *tuple-based* and *time-based* sliding windows. The former defines the window boundary based on the number of tuples, also referred to as the count-based window semantic, and the latter uses time to delimit the window. We present our approach based on tuple-based sliding windows, although there is no technical limitation for applying our approach to time-based sliding windows. We denote a two-way window θ-join as $W_R \bowtie_\theta W_S$, where W_R and W_S are the sliding windows of streams R and S, respectively. The join result contains all pairs of the form (r, s) such that $r \in W_R$ and $s \in W_S$, where $\theta(r, s)$ evaluates to *true*. A join operator processes a tuple r arriving at stream R as follows. (1) Lookup r in W_S to determine matching tuples and propagate the results into the output stream. (2) Delete expired tuples from W_R. (3) Insert tuple r into W_R. The cost of each step depends on the choice of the join algorithm and index data structure used. To simplify the time complexity analysis for different join implementations, we assume that the lengths of the sliding windows of both streams, R and S, are identical, denoted by w. Additionally, we ignore the cost of the sliding window update in our analysis since it is identical when using different join algorithms and indexing approaches.

2.2 Index-Based Window Join

IBWJ accelerates window lookup by utilizing an index data structure. Although maintaining an extra data structure along the sliding window increases the update cost, the performance gain achieved during lookup offsets this extra cost and results in higher overall throughput. The general idea

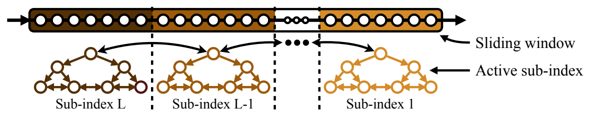

Figure 2: Chained index.

of IBWJ is illustrated in Figure 1. Tuples in W_R and W_S are indexed into two separate index structures called I_R and I_S, respectively. Upon the arrival of a new tuple r into stream R, IBWJ searches I_S for matching tuples. In addition, I_R must be updated based on the changes in the sliding window. Here, we examine IBWJ using B$^+$-Tree, chained index and context-insensitive partitioning.

2.2.1 *IBWJ using B$^+$-Tree.*

We now derive the time complexity of IBWJ based on B$^+$-Tree. Let H_b be the height of the B$^+$-Tree storing w records ($H_b \approx \log_{f_b}^{w}$). The join algorithm processes a given tuple r from stream R as follows. (1) Search I_S to reach a leaf node ($H_b \cdot \lambda_b^s$); then, linearly scan the leaf node to determine all matching tuples ($\sigma_s \cdot \tau_c$). (2) Delete the expired tuple from I_R ($H_b \cdot \lambda_b^d$). (3) Insert the new tuple, r, into I_R ($H_b \cdot \lambda_b^i$).

2.2.2 *IBWJ using Chained Index.*

Lin et al. [24] and Ya-xin et al. [43] proposed *chained index* to accelerate stream join processing. The basic idea of chained index is to partition the sliding window into discrete intervals and construct a distinct index per each interval. Figure 2 depicts the basic idea of chained index. As new tuples arrive into the sliding window, they are inserted into the active subindex until the size of the active subindex reaches its limit. When this situation occurs, the active subindex is archived and pushed into the subindex chain, and an empty subindex is initiated as a new active subindex. Using this method, there is no need to delete expired tuples incrementally; rather, the entire subindex is released from the chain when it expires.

We now derive the time complexity of IBWJ when both I_R and I_S are set to a chain index of length L ($L \geq 2$) and all subindexes are B$^+$-Trees. Let H_c be the height of each subindex ($H_c \approx H_b - \log_{f_b}^{L}$; we also considered the height of the active subindex being equal to that of archived subindexes to simplify the equations). The join algorithm processes a given tuple r from stream R as follows. (1) Search all subindexes of I_S to their leaf nodes ($L \cdot H_c \cdot \lambda_b^s$) and linearly scan leaf nodes to find matching tuples and filter out expired tuples during the scan. The number of expired tuples that need to be removed from the result set is $\sigma_s/(2 \cdot (L-1))$ on average. (2) Check whether the latest subindex of I_R is expired and discard the entire subindex. The cost of this step is negligible, and we consider it to be zero. (3) Insert the new tuple, r, into the active subindex of I_R ($H_c . \lambda_b^i$).

Comparing the cost of the index operations using chained index and B$^+$-Tree indicates that using chained index to index sliding windows is more efficient in terms of index update costs than using a single B$^+$-Tree, whereas range queries are

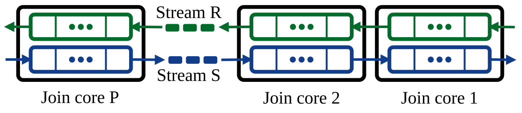

Figure 3: Low-latency handshake join.

more costly using chained index because it needs to search multiple individual subindexes.

2.2.3 IBWJ using Round-Robin Partitioning. A group of parallel stream join solutions, such as handshake join [34], SplitJoin [28] and BiStream [24], are based on context-insensitive partitioning. In all these mentioned approaches, a sliding window is divided into disjoint partitions using round-robin partitioning which is based on the arrival order of tuples rather than tuple values, and each join core is associated with a single window partition. To accelerate the lookup operation, each thread may maintain a local index for its associated partition. Because indexes are local to each thread, there is no need for a CC mechanism to access indexes. In fact, the parallelism in these approaches is achieved by dividing a tuple execution task into a set of independent subtasks rather than utilizing a shared index data structure and distributing tuples among threads. As a drawback of approaches based on context-insensitive partitioning, it is required to have all joining threads available to generate the join result of a single tuple because each thread can only generate a portion of the join result.

Here, we explain the cost of IBWJ using the low-latency variant of handshake join (LHS) employing P threads. Figure 3 illustrates the join-core arrangement and the flow of streams in LHS. In LHS, join cores are linked as a linear chain such that each thread only communicates with its two neighbors, and data streams R and S propagate in two opposite directions. In the original handshake join, tuples arrive and leave each join core in sequential order, and tuples may have to queue for a long period of time before moving to the next join core. This results in significant latency in join result generation and in higher computational complexity because all tuples are required to be inserted and deleted from each local index. In LHS, however, tuples are fast forwarded toward the end of the join core chain to meet all join cores faster. Moreover, each tuple is only indexed by a single join core, which is assigned in a round-robin manner. Consequently, LHS results in higher throughput and lower latency than the original handshake join.

We now derive the time complexity of the index operations required to process a single tuple using round-robin partitioning with P join cores. Let all join cores use B$^+$-Tree as local indexes and H_p be the height of each local index ($H_p \approx H_b - \log_{f_b}^{P}$). The cost of processing a given tuple r from stream R is as follows. (1) Tuple r is propagated among all join cores, and all cores search their local I_S until the leaf nodes ($P \cdot H_p \cdot \lambda_b^s$) and linearly scan leaf nodes to find matching tuples ($\sigma_s . \tau_c$). (2) The join core assigned to index tuple r deletes the expired tuple from its I_R ($H_p \cdot \lambda_b^d$). (3) The same join core as in Step 2 inserts the new tuple, r, into its I_R ($H_p \cdot \lambda_b^i$).

Comparing the cost of the index operations using round-robin partitioning with the cost of IBWJ using B$^+$-Tree results in the following: Using round-robin partitioning is more efficient for inserting or deleting a tuple from a sliding window than using a single B$^+$-Tree because the heights of the local indexes for each partition are less than a single B$^+$-Tree indexing w tuples ($H_p < H_b$). However, because it is necessary to search multiple local indexes using round-robin partitioning to find matching tuples, using a single B$^+$-Tree is more efficient in terms of range querying. Generally, as the number of join cores increases, the total cost of searching local indexes using round-robin partitioning also increases, which is a consequence of context-insensitive window partitioning. This redundant index search limits the efficiency of approaches based on round-robin partitioning in the application of IBWJ.

3 CONCURRENT WINDOW INDEXING

In this section, we present the design of our indexing data structures for join processing.

3.1 Overview

We propose a novel two-stage indexing mechanism to accelerate parallel stream joins by combining two existing techniques, *delta merging* and *data partitioning*, resulting in a highly efficient indexing solution for both single- and multithreaded sliding window indexing. Our indexing solution consists of a mutable component and an immutable component. The mutable component is an insert-efficient indexing data structure in which all the new tuples are initially inserted. The immutable component is a search-efficient data structure where updates are applied using delta merging. Utilizing the strength of each indexing component and a coarse-grained tuple disposal method, our two-stage data structure results in more efficient sliding window indexing compared with a single-component indexing data structure. Moreover, we extend our indexing solution by splitting the mutable component into multiple mutable partitions, where partitions are assigned to disjoint ranges. Consequently, operations on different value ranges can be performed concurrently. This technique enables our indexing solution to leverage the distribution of queries to support efficient task parallelism with a lightweight CC mechanism. In this section, we first study the effect of delta merging in the application of sliding window indexing, and then we extend the delta merging method with index partitioning to support parallel sliding window indexing.

In this work, we use two different B$^+$-Tree designs that have distinct performance characteristics. The first design is the classic B$^+$-Tree design, where each node explicitly stores the references to its children. This design, which we simply refer to as *B$^+$-Tree*, supports efficient incremental updates. In contrast, as an immutable data structure, B$^+$-Tree nodes can be arranged into an array in a breadth-first fashion. In this representation, given a node position, it is possible to retrieve the location of its children implicitly without needing to store actual references. By eliminating child references, more space is available in inner nodes for keys, and it is feasible to achieve a higher fan-out and decrease the tree depth. Therefore, lookup operations in this design, which we call *immutable B$^+$-Tree*, are faster than in the classic design based on node referencing. As a drawback, it is inefficient to perform individual updates in an immutable B$^+$-Tree since the entire tree must be reconstructed; however, this type of access is not required in our use of the index.

Throughout this paper, λ^s_{ib} denotes the time complexity of search at each node of the immutable B$^+$-Tree, and f_{ib} denotes the inner node fan-out of immutable B$^+$-Tree.

3.2 In-memory Merge-Tree

We now describe our *in-memory merge tree* (IM-Tree), which is designed to accelerate sliding window indexing. IM-Tree consists of two separate indexing components (T_I and T_S). T_I is a regular B$^+$-Tree that is capable of performing individual updates, and T_S is an immutable B$^+$-Tree that is only efficient for bulk updates. All new tuples are initially indexed by T_I. When the size of T_I reaches a predefined threshold, the entire T_I is merged into T_S, and simultaneously, all expired tuples in T_S are discarded. The merging threshold is defined as $m \times w$, where m is a parameter between zero and one ($0 < m \leq 1$), referred to as the *merge ratio.* To query a range of tuples, it is necessary to search both components, T_I and T_S, separately. Additionally, it is necessary to filter out expired tuples of T_S from the result set. When a tuple expires, it is flagged in the sliding window as expired but not eliminated. To drop expired tuples from the index search results, every result tuple is checked in the sliding window to determine whether it is flagged as expired. At the end, all expired tuples are eliminated from both the sliding window and the index data structure during the merge operation. Therefore, we must store an additional $w \times r$ tuples in the sliding window to use IM-Tree.

Both chained index and IM-Tree utilize a coarse-grained tuple disposal technique to alleviate the overhead of tuple removal, but the tuple disposal techniques differ between these indexing approaches. Chained index disposes of an entire subtree, whereas IM-Tree eliminates expired tuples periodically during the merge operation. The periodic merge enables IM-Tree to maintain all indexed tuples in only two index components and to provide better search performance than chained index.

Although both LSM-Tree [29] and IM-Tree are multicomponent indexing solutions that use the delta updating mechanism to transfer data among their components, the two data structures are designed differently to tackle distinct problems. Components in LSM-Tree are configured to be used in different storage media, and LSM-Tree applies delta updating to alleviate the cost of write operations in low-bandwidth storage media. In contrast, IM-Tree consists of two in-memory components specialized for different operations, and IM-Tree applies periodic merges to enhance the performance of range queries. Moreover, LSM-Tree is based on incremental merging between its components, which is not applicable to immutable data structures such as the immutable B$^+$-Tree used in our IM-Tree.

3.2.1 IBWJ using IM-Tree. Let H_I and H_S be the heights of T_I and T_S, respectively. The time complexity of processing a tuple s arriving at stream S for IBWJ using IM-Tree is as follows. (1) Search both T_I and T_S of the opposite stream to the leaf nodes ($H_I \cdot \lambda^s_b + H_S \cdot \lambda^s_{ib}$) and perform a linear scan of the leaf node to determine matching tuples ($\sigma_s \cdot \tau_c$) and filter out expired tuples ($\sigma_s \cdot \tau_c \cdot \frac{m}{2}$). (2) Tuples in IM-Tree are deleted in a batch during a T_I and T_S merge. Let M be the time complexity of the merge; then, the average cost per tuple is $M/(m \cdot w)$. (3) Insert the new tuple into the index of stream S ($H_I \cdot \lambda^i_b$).

The stepwise comparison between the window join using B$^+$-Tree and IM-Tree is controlled by the merge ratio m. Assigning a proper value for m is subject to various trade-offs. A late merge creates a larger T_I on average and results in a more expensive insert and search of T_I. Additionally, it increases the average number of expired tuples in T_S and results in an inefficient lookup in T_S. Meanwhile, merge operations are costly, and overdoing such operations results in a significant performance loss. Generally, increasing the value of m causes the costs of Steps 1 and 3 to increase and the cost of Step 2 to decrease.

The memory space required for IM-Tree consists of three parts, T_I, T_S and the merge buffer. T_I is a B$^+$-Tree that stores at most $r \times w$ tuples. T_S is a immutable B$^+$-Tree that stores w tuples. Moreover, we must maintain a buffer of size w tuples needed for merging T_I and T_S in each merge phase.

3.3 Partitioned In-memory Merge-Tree

Partitioned in-memory merge tree (PIM-Tree) is an extended variant of IM-Tree that is designed to address the challenges of parallel sliding window indexing. Similar to IM-Tree, PIM-Tree is also composed of two components in which recently inserted tuples are periodically merged into a lookup-efficient index. In fact, the key difference is in the design of the insert-efficient component T_I. Rather than using a single B$^+$-Tree for all incoming tuples, we opt to use a set of B$^+$-Trees that are associated with disjoint tuple value ranges. To provide a uniform workload among trees, these ranges periodically adapt to the distribution of values in the sliding window. Each B$^+$-Tree is associated with a lock that allows only a single thread to access the tree to handle parallel updates and lookups. Unlike approaches that target resolving concurrency at the tree node level, such as Bw-tree [23] or B-link [20], parallelism in PIM-Tree is based on concurrent operations over disjoint partitions and relies on the distribution of incoming tuples. An advantage of our approach is that the routines for performing operations are as efficient as those of the single-threaded approach, and their only overhead is to obtain a single lock per each tree traversal.

3.3.1 PIM-Tree Structure. Figure 4 provides an overview of the PIM-Tree structure. PIM-Tree consists of two separate components, T_S and T_I. T_S is an immutable B$^+$-Tree; it is similar to our IM-Tree, which stores static data. T_I represents a set of *subindexes* named $B_0, .., B_n$ attached to T_S at depth D_I (*insertion depth*), where each B_i is associated with the same range of values as the i^{th} node of T_S at the insertion depth. Each B_i is an independent B$^+$-Tree, where the tail leaf node of each B_i $(0 \leq i < n)$ is connected to the head leaf node of the successor B$^+$-Tree (B_{i+1}) to create a single sorted linked list of all elements in T_I.

To insert a new record, the update routine first searches T_S until the depth of D_I to identify the matching B_i that is associated with the range that includes the given value. Then, the routine inserts the record into B_i using the B$^+$-Tree insert algorithm. Similar to IM-Tree, the two components of PIM-Tree need to be periodically merged for maintenance. This maintenance occurs when the total number of tuples in T_I equals $m \times w$. Merging eliminates expired tuples in T_S and arranges the remaining tuples to be combined with those from T_I into a sorted array that is taken as the last level of the new T_S. Subsequently, T_S is built from the bottom up, and every B_i is initialized as an empty B$^+$-Tree.

3.3.2 IBWJ Using PIM-Tree. Let H'_I be the average height of $B_i, 0 \leq i \leq n$. The join algorithm processes a given tuple r from stream R as follows. (1) Search the index of stream S to identify matching tuples, which requires first searching T_S ($H_S \cdot \lambda^s_{ib}$) and the corresponding B_i ($H'_I \cdot \lambda^s_b$) to the leaf nodes and then performing a leaf node scan to determine matching tuples and filter out expired tuples ($\sigma_s \cdot \tau_c \cdot (1+m/2)$). (2) Similar to IM-Tree, tuples are deleted in a batch during the merge of T_I and T_S; thus, the average cost per tuple is $M'/(m \cdot w)$, where M' is the cost of merging T_I and T_S in PIM-Tree. (3) Insert the new tuple, r, into T_I, which requires first traversing T_S to depth D_I ($D_I \cdot \lambda^s_{ib}$) and then inserting the tuple into the corresponding B_i ($H'_I \cdot \lambda^i_b$).

In terms of memory footprint, PIM-Tree requires almost the same amount of memory space as IM-Tree. The amounts

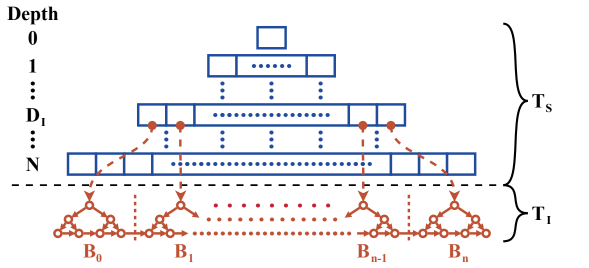

Figure 4: Structure of PIM-Tree (blue and red sections are T_S and T_I components, respectively).

of memory required for T_S and the merge buffer are identical between PIM-Tree and IM-Tree. Moreover, considering that leaf nodes take up the most space in B$^+$-Tree, the memory space difference between T_I in IM-Tree and PIM-Tree is negligible in comparison with the size of the entire tree.

Comparing the costs of IBWJ using IM-Tree and PIM-Tree, we obtain the following. Searching in PIM-Tree is faster because the average height of a subindex in PIM-Tree is less than T_I in IM-Tree. The costs for merging T_I and T_S in both trees are almost identical ($M = M'$), and consequently, the overall cost of tuple deletion is the same in both trees. The insertion costs in PIM-Tree and IM-Tree are controlled by the number of tuples in T_I. Let the number of tuples in T_I be represented by $|T_I|$. For $|T_I| = 0$ (after merge), the constant overhead of traversing T_S to depth D_I in PIM-Tree is dominant and results in slower insertion in PIM-Tree. As $|T_I|$ increases, the cost of insertion in IM-Tree increases faster and eventually surpasses the insertion cost in PIM-Tree.

3.3.3 Concurrency in PIM-Tree. To protect the PIM-Tree structure during concurrent indexing, each subindex (B_i) is associated with a lock that coordinates the accesses of the threads to the subindex. Moreover, a searching thread may move from a B_i to its successor (B_{i+1}) during the leaf node scan to determine matching tuples. To address this issue, the last leaf node of each B_i is flagged such that the searching thread recognizes the movement from one subindex to another. In this case, the searching thread releases the lock and acquires the one associated with the successor. Traversing T_S is completely lock-free since its structure never changes, and there is no need for a CC mechanism to avoid race conditions.

In the case of a fixed tuple value distribution, the insert operations are spread uniformly across subindexes, even though the tuple value distribution is skewed. The reason is that B$^+$-Tree nodes are naturally adapting to the indexed values such that the subtrees of the two inner nodes at the same depth have almost an equal number of indexed values. Because T_I's subindexes are adjusted according to T_S's inner nodes, the load is uniformly distributed among subindexes regardless of the value distribution. However, when the distribution changes, the range assignment is no longer optimal and causes skew among subindexes in the insert operation.

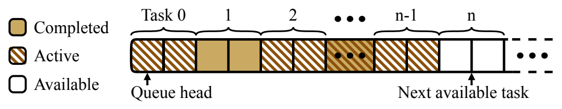

Figure 5: Shared task queue with task size of 2.

Because the workload distribution among subindexes is readjusted in every merge phase, PIM-Tree will ultimately recover to normal performance in the first merge phase after the data distribution stops changing.

4 PARALLEL STREAM JOIN USING SHARED INDEXES

In this section, we present our parallel window join algorithm, which addresses the challenges of using shared indexes in a multithreaded setting. During concurrent join, tuples might be inserted into indexes in an order different from their arrival order depending on the threads' scheduling in the system. We design a join algorithm that is aware of the indexing status of tuples to avoid duplicated or missing results. Moreover, our join algorithm is based on an asynchronous parallel model, which enables threads to dynamically join or leave the operator depending on system load.

4.1 Concurrent Stream Join Algorithm

Our parallel join algorithm processes incoming tuples in four steps: (1) task acquisition, (2) result generation, (3) index update, and (4) result propagation.

Task acquisition – A task represents a unit of work to schedule, which is a set of incoming tuples. The task size is the number of tuples assigned to a thread per each task acquisition round, which determines the trade-off between maximizing throughput and minimizing response time. Large tasks reduce scheduling and lock acquisition overhead but simultaneously increase system response time, whereas small tasks result in the opposite. In our join algorithm, tasks are distributed among threads based on dynamic scheduling; thus, a thread is assigned with a task whenever the thread is available. This method enables our join algorithm to utilize an arbitrary number of threads and not stall because threads are unavailable.

We arrange incoming tuples into a shared work queue according to their arrival order, regardless of which stream they belong to; we protect the access to this queue using a shared mutex. Each tuple in the work queue is assigned a status flag: *available* indicates that the tuple is ready to be processed but not yet assigned to any thread, *active* indicates that the tuple is assigned to a thread but the join results are not ready, and *completed* indicates that processing of the tuple is completed and the join results are ready but the results are not propagated. When a tuple arrives in the queue, its status is initialized to available, and a completed tuple remains in the work queue until the results of previous tuples are propagated into the output stream. Figure 5 illustrates

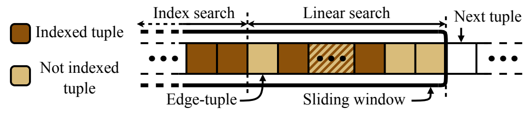

Figure 6: Sliding window during parallel stream join.

the status of the work queue during a window join with a task size of 2.

During a concurrent stream join, sliding windows must store all tuples that are required to process active tuples of the opposite stream, which generally results in windows larger than w. In the case of a time-based sliding window, it is possible to filter out unrelated tuples using timestamps; however, for count-based sliding windows, it is necessary to record the boundaries of the opposite window at the point in time when a tuple is assigned to a thread. We refer to these boundaries as t_l (latest tuple) and t_e (earliest tuple). When a thread acquires a task, it changes the status of the tuples to active and saves t_l and t_e for each tuple.

Result generation – To avoid duplicate or missing results, we keep references to the earliest nonindexed tuple of each sliding window, referred to as the *edge tuple*. This tuple declares that all tuples before it are already indexed, whereas the statuses of the subsequent tuples are undetermined. When a thread starts to process a tuple, it stores the position of the edge tuple in a local variable since the value might be updated during processing. Using an old value of the edge tuple might increase the computational cost slightly, but it is safe in terms of result correctness. The lookup algorithm determines matching tuples in two steps. First, it queries the index for matching tuples and filters out those after the edge tuple or before t_l. Second, it linearly searches the sliding window from the edge tuple to t_e and adds any results to the previously found results. Figure 6 illustrates the sliding window during the join operation. When a thread finishes processing a tuple, it stores the results in shared memory and updates the task status to *completed* in the shared queue but does not yet propagate the results into the output stream at this step.

Index update – After a thread generates the join results for a tuple, it inserts the tuple into the index and marks the tuple in the sliding window as indexed. Subsequently, the thread attempts to update the edge tuple accordingly. To avoid a race condition, a shared mutex coordinates write accesses to the edge tuple. Using a test-and-set operation, the thread checks whether the mutex is held by another thread. If so, it avoids the edge tuple update and continues to the next step. Otherwise, it increments the edge tuple to the next nonindexed tuple in the sliding window and releases the mutex.

Result propagation – In the final step, a thread attempts to propagate the results of completed tuples. Similar to the edge tuple update routine, a shared mutex coordinates threads

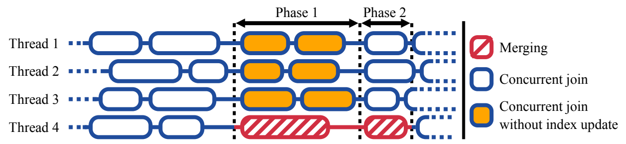

Figure 7: Nonblocking merge.

during result propagation. The thread checks the status of the mutex. In the case that the mutex is already held by another thread, the thread skips this step and begins to process another task. Otherwise, it verifies whether the results for the tuple at the work queue head are completed. If so, it propagates the results into the output stream and removes the tuple from the work queue. If the tuple is not completed, the thread does not propagate the results of any other completed tuple in order to ensure that results are propagated into the output stream according to the tuples' arrival order. This routine is repeated until the status of the tuple at the work queue head is either active or available. Finally, the thread releases the mutex and starts to process another task.

4.2 Nonblocking Merge and Indexing

Performing merging as a blocking operation negatively impacts system availability and latency, which are both often critical concerns for stream processing applications. To address this challenge, we propose a nonblocking merge method. Our approach enables the stream join processing threads to continue the join without significant interruption during merge processing. Figure 7 illustrates the overall scheme of performing a nonblocking merge. The operation consists of two phases: first, creating a new PIM-Tree, and second, applying pending updates.

Whenever merging is needed, a thread called the *merging thread* is assigned to perform the merge operation. At the beginning of each stage, the merging thread blocks the assignment of new tasks until all active threads finish their currently processing tasks. During the first phase, the merging thread creates a new PIM-Tree without modifying the previous index tree. Concurrently, other threads resume performing tasks without an index update. When the merging thread finishes creating the updated PIM-Tree, it starts the next phase. At the beginning of the second phase, the merging thread swaps the old index with the new one before it unblocks the task assignment process. During the second phase, the merging thread applies pending updates and other threads begin to perform the join operation with index update. When the pending updates are finished, the merging thread leaves the merge operation and begins to perform the join operation.

During the first phase of a nonblocking merge, the index data are not updated; therefore, the position of the edge tuple does not change during this phase. Consequently, the linear search in the nonindexed portion of the sliding window becomes more expensive.

5 EVALUATION

In this section, we present a set of experiments to benchmark the efficiency of the approaches introduced in this paper and empirically determine the corresponding parameters, such as merge ratio and insertion depth. Moreover, we study the influence of join selectivity and skewed value distribution on the performance of our parallel window join design. As the query workload, we use the following micro-benchmark where two streams, R and S, are joined via the following band join.

```
SELECT * FROM R, S
WHERE ABS(R.x - S.x) <= diff
```

The join attributes ($R.x$ and $S.x$) are assumed to be random integers generated according to a uniform distribution and the input rates of streams R and S are symmetric unless otherwise stated. Each tuple consists of a 4-byte key and 12 bytes of payload. Because our indexing data structure only stores keys and references to a sliding window, neither the footprint nor the indexing performance of the data structures evaluated in this paper are influenced by tuple size. However, the tuple size might influence the parallel join algorithm by increasing the cost of moving tuples from the work queue to the sliding window. This cost is not specific to our parallel join algorithm and it would equally influence any other join algorithm.

Because we evaluate each experiment for different window lengths (w), considering a fixed value for *diff* results in various join match rates (i.e., the match rate of band join with $w = 2^{25}$ will be 2^{15} times higher than that with $w = 2^{10}$), which influences the overall join performance. For a more comprehensive comparison, the value of *diff* is adjusted according to the window length such that the match rate (σ_s) is always two except for the one that exclusively studies the influence of join selectivity. We used two forms of band join: two-way join and self-join. In the former, R and S are two distinct streams, and in the latter, an identical stream is used as both R and S. The experiments are generally based on two-way join, except for those where we explicitly declare that self-join is used. To cover a large range of window sizes, we opted to use a logarithmic scale of base two in our experiments. However, there is no technical limitation in using our indexing mechanism for any arbitrary sizes.

We evaluate our approaches on an octa-core (16 CPU threads, hyperthreading enabled) Intel Xeon E5-2665. For all multithreaded experiments, we utilized all 16 threads unless otherwise stated. We employ the STX-B$^+$-Tree implementation, which is a set of C++ template classes for an in-memory B$^+$-Tree [5], and we used our own CSS-Tree implementation as immutable B$^+$-Tree [31]. [1]

[1]Due to space limitations, we provide an extended technical report with additional experimental results in [37].

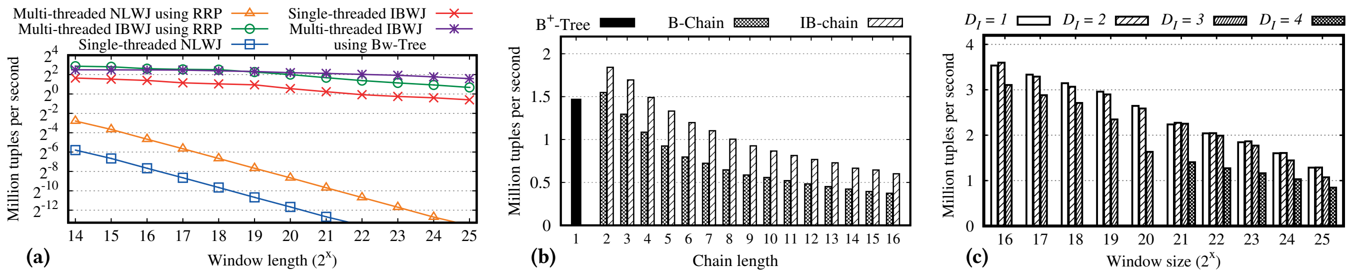

Figure 8: a) Performance evaluation of multithreaded window join using round-robin partitioning (RRP). b) Throughput comparison of IBWJ using chained index and B⁺-Tree. c) Throughput vs. insertion depth for single-threaded IBWJ using PIM-Tree.

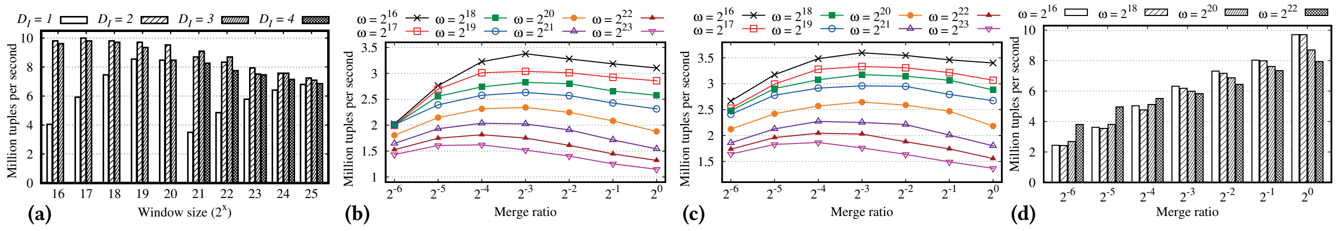

Figure 9: a) Throughput vs. insertion depth for parallel IBWJ using PIM-Tree. b) Throughput vs. merge ratio for IBWJ using IM-Tree. c) Throughput vs. merge ratio for IBWJ using PIM-Tree. d) Throughput vs. merge ratio for parallel IBWJ.

5.1 Comparison of Existing Approaches

Round-robin window partitioning – The purpose of this experiment is to study the efficiency of round-robin partitioning-based approaches, such as low-latency handshake join, SplitJoin and BiStream, in the application of index-accelerated stream join. We evaluate five implementations of the window join: (1) single-threaded nested-loop window join (NLWJ), (2) multithreaded NLWJ based on round-robin partitioning, (3) single-threaded IBWJ using B⁺-Tree, (4) multithreaded IBWJ based on round-robin partitioning, and (5) multithreaded IBWJ using Bw-tree. Figure 8a presents the results for varying window sizes.

Comparing the join algorithms, we observe that NLWJ is more vulnerable to the sliding window size because its performance linearly decreases as the window size increases. In contrast, the performance of IBWJ is less sensitive to the sliding window size. Multithreaded join using round-robin partitioning improves the performances of NLWJ and IBWJ by factors of 8 and 2.5, respectively. This result implies that although approaches based on round-robin window partitioning are effective for NLWJ, these approaches cannot efficiently exploit the computational power of multicore processors for IBWJ.

Moreover, the performance result of parallel IBWJ using Bw-Tree indicates that the efficiency of concurrent operations in Bw-Tree improves as the size of Bw-Tree increases. The larger the indexing tree, the lower is the probability of accessing the same node by different threads at the same time; consequently, the multithreading efficiency increases. For the smallest sliding window size ($w = 2^{14}$), parallel IBWJ using Bw-Tree results in 65% lower throughput than parallel IBWJ using round-robin partitioning, but for the largest window size ($w = 2^{25}$) evaluated, parallel IBWJ using Bw-Tree outperforms the round-robin-based method and results in 75% higher throughput.

Chained index – Figure 8b shows the throughput of IBWJ using chained index [24] for varying chain lengths in comparison with B⁺-Tree ($w = 2^{20}$). For this experiment, we set the insertion depth to one ($D_I = 1$) and merge ratio (r) to 1/8. We propose and evaluate two different designs for chained index, referred to as *B⁺-Tree chain (B-chain)* and *Immutable B⁺-Tree chain (IB-chain)*. In the former design, all subindexes are B⁺-Trees, including the active subindex (the one where newly arriving tuples are inserted) and all archived subindexes. In the latter design, only the active subindex is a B⁺-Tree, and before archiving an active subindex, it is converted into an immutable B⁺-Tree; thus, all archived subindexes are immutable B⁺-Trees.

We observe that the IB-chain results in 50% higher throughput than the B-chain on average, which indicates that the immutable B⁺-Tree vastly outperforms the regular B⁺-Tree for search queries in this scenario. For both the B-chain and IB-chain, the shortest chain length, which is two, results in the best throughput. However, the performance noticeably decreases when the chain length increases. The main drawback of chained index is the higher search complexity, which increases almost linearly with the chain length. Although the index chain reduces the overhead of tuple removal using coarse-grained data discarding, the higher search overhead degrades its overall performance.

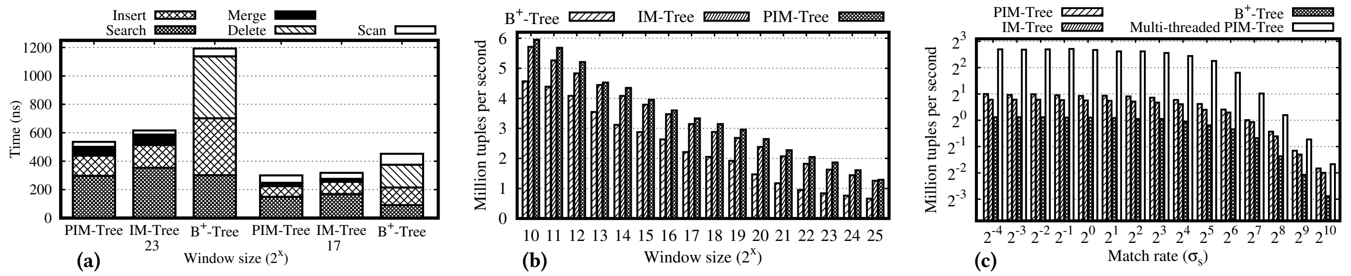

Figure 10: a) Cost comparison of the different steps of IBWJ for a single tuple using various indexing data structures. b) Performance comparison of single-threaded IBWJ using different indexing data structures. c) Throughput vs. match rate for IBWJ.

5.2 IBWJ using PIM-Tree and IM-Tree

Insertion depth – In this experiment, we study the impact of the insertion depth (D_I) on the performance of PIM-Tree. Increasing D_I results in smaller subindexes (B_is), which accelerates the operation on subindexes, and it simultaneously increases the overhead of searching T_S to find the corresponding B_i. Figures 8c and 9a show the throughputs of single-threaded and parallel IBWJ, respectively, using PIM-Tree for different D_Is ranging from one to four, considering that the root node is at a depth of zero. For the window sizes of 2^{16} to 2^{19}, there are only four levels of inner nodes (including the root node); thus, the maximum feasible D_I is three. The results for $D_I = 1$ reveal that the number of inner nodes at depth D_I highly influences the performance of parallel IBWJ. If the number of subindexes in T_I (which is equal to the number of inner nodes at depth D_I) is not sufficient, then the performance significantly decreases due to the high partition locking congestion. From $w = 2^{16}$ to 2^{20}, the system throughput rapidly increases since the number of inner nodes at $D_I = 1$ also increases. At $w = 2^{21}$, the number of inner nodes at $D_I = 1$ decreases since the tree depth is incremented by one, which also causes a decrease in the IBWJ throughput. For larger values of D_I (three and four), the IBWJ throughput does not improve, which suggests that the multithreading is no longer bounded by the number of subindexes. For the case of single-threaded IBWJ, the achieved throughput for different D_Is is less dependent on the window size. However, setting D_I to the highest feasible value results in a higher overhead for searching T_S and lowers the overall performance.

Merge ratio (m) – To determine the empirically optimal merge ratio for IM-Tree and PIM-Tree, we conduct an experiment for each data structure. Figures 9b and 9c illustrate the throughputs of single-threaded IBWJ using IM-Tree and PIM-Tree, respectively, with merge ratios ranging from 2^{-6} to 1. The results for both data structures follow a similar pattern, but the average throughput employing PIM-Tree is higher than that using IM-Tree. Additionally, the system does not perform efficiently for either very low or very high values of the merge ratio. This underperformance is a consequence of the excessive overhead imposed by the frequent merge when the merge ratio is set very low and by the inefficient insert and search operations when the merge ratio is set very high. The results suggest that the choice of the merge ratio is more influential for smaller sliding windows, and the empirical optimal ratio is not identical for all window sizes. Over the largest evaluated sliding window (2^{23}), setting the merge ratio to $1/2^4$ results in the highest throughput, whereas for the smallest one (2^{16}), $1/2^3$ is the best merge ratio.

Figure 9d illustrates the throughput of the parallel IBWJ using PIM-Tree for varying merge ratios ranging from 2^{-6} to 1. In contrast to the single-threaded implementation, setting the merge ratio to the highest value always results in the best performance in the multithreaded setting, regardless of the window size. This result indicates that the cost of merge operations during a parallel window join is higher than the cost in a single-threaded setup. Hence, minimizing the number of merges results in the highest throughput. We also observe that the choice of the merge ratio is more influential for smaller window sizes. Henceforth, we set the value of the merge ratio for the multithreaded setup to one.

B$^+$-Tree vs. IM-Tree vs. PIM-Tree – In this experiment, we compare the performances of IBWJ using B$^+$-Tree, IM-Tree and PIM-Tree. For a more comprehensive comparison, we divide the process of finding matching tuples into two steps: traversing the index tree for the tuple with the lowest value, referred to as *searching*, and linearly checking tuples in leaf nodes, referred to as *scanning*. For each data structure, we measure the costs of the different steps of performing IBWJ, including insert, delete, search, scan, and merge. Figure 10a shows the results for sliding window sizes of 2^{17} and 2^{23}.

The merging overhead is almost identical for both IM-Tree and PIM-Tree, and it constitutes 7% and 11% of the total processing for 2^{17} and 2^{23} windows, respectively. Regarding the tuple insertion performance, PIM-Tree and IM-Tree perform nearly identically, and they are 1.5 and 2.6 times faster than B$^+$-Tree for 2^{17} and 2^{23} windows, respectively. For the smaller window size (2^{17}), searching in B$^+$-Tree is 75% faster than searching in IM-Tree and PIM-Tree. However, for the larger window size (2^{23}), the search performances corresponding

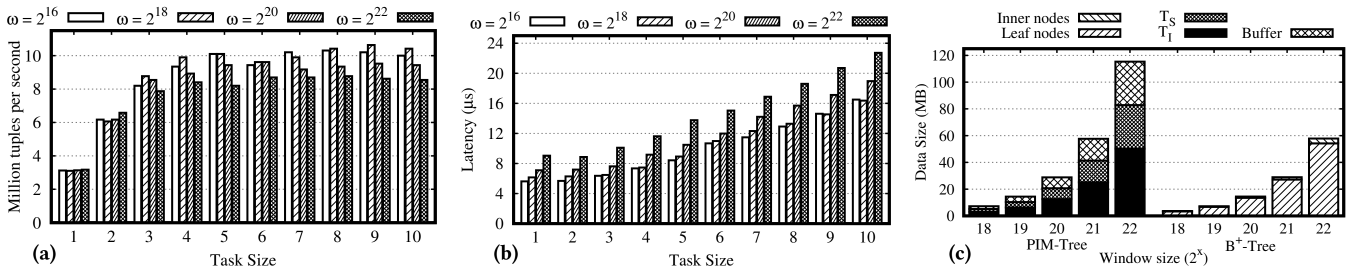

Figure 11: a) Throughput vs. task size for parallel IBWJ using PIM-Tree. b) Latency vs. task size for parallel IBWJ using PIM-Tree. c) Memory footprint comparison of B$^+$-Tree and PIM-Tree.

to PIM-Tree and B$^+$-Tree are nearly identical, and both are slightly faster than IM-Tree.

Figure 10b presents the throughput of single-threaded IBWJ using B$^+$-Tree, IM-Tree, and PIM-Tree for varying window sizes. We observe that employing PIM-Tree and B$^+$-Tree results in the best and the worst performances, respectively. Considering IBWJ using B$^+$-Tree as the baseline, average improvements in system performance of 50% and 63% in magnitude are achieved by employing IM-Tree and PIM-Tree, respectively.

Match rate (σ_s) – Figure 10c shows the throughputs of four different implementations of IBWJ for the window size of 2^{20} and match rates varying from 2^{-4} to 2^{10}. These implementations are three single-threaded IBWJ using B$^+$-Tree, IM-Tree and PIM-Tree and one multithreaded IBWJ using PIM-Tree. The join performance varies negligibly for the match rates between 2^{-4} and 2^4, which indicates that the join performance in this range is bounded by index traversing rather than the linear leaf node scans. As the match rate increases beyond 2^4, the join performance noticeably decreases for all implementations. This result implies that for higher match rates, i.e., $2^5 \leq \sigma_s \leq 2^{10}$, the join performance is bounded by system memory bandwidth due to extensive leaf node scans. Consequently, multithreading loses its advantage for IBWJ with high selectivities, and its performance becomes closer to that of the single-threaded implementations. Additionally, the result indicates that single-threaded IBWJ using IM-Tree and PIM-Tree for joins with high selectivity results in better performance than using B$^+$-Tree, which is because of the more efficient leaf node scan in immutable B$^+$-Tree (T_S) than in regular B$^+$-Tree.

Task size – In this experiment, we study the influence of the task size on our parallel window join algorithm. Increasing the task size decreases the overhead of task acquisition while simultaneously increasing the system latency (task processing time). Figures 11a and 11b illustrate the performance of IBWJ using PIM-Tree over different task sizes ranging from 1 to 10 in terms of throughput and latency, respectively. Increasing the task size to four steadily improves the performance, which suggests that very small task sizes lead to significant task scheduling overhead. For task sizes from five to eight, a minor improvement is achieved, and for task sizes larger than eight, the performance does not significantly vary. The evaluation results shown in Figure 11b indicate that the task size greatly influences the system latency: increasing the task size leads to higher latencies. Additionally, we observe that the latency of parallel IBWJ is higher for larger sliding windows. As the window size increases, the PIM-Tree merge becomes more costly because it leads to longer linear window scans during nonblocking merge and consequently causes higher latency. In the remainder of the evaluation, we use tasks of size eight.

Memory consumption – In this experiment, we compare the memory footprint of IBWJ using PIM-Tree and B$^+$-Tree. We assume that the merge ratio is one ($r = 1$) in this experiment, such that T_I and the sliding window for IBWJ using PIM-Tree is of the largest possible size. The memory footprint of IBWJ consists of two components, the indexing data structure and sliding window. Figure 11c compares the memory space required for different components of PIM-Tree and B$^+$-Tree storing varying numbers of elements. Each element is a pair of 4 bytes for the key and 4 bytes for the sliding window reference. The storage required for PIM-Tree consists of the search-efficient component (T_S), the insert-efficient component (T_I), and a buffer that is required during merge. The results reveal that the space required for PIM-Tree is almost double the space required for B$^+$-Tree, regardless of window size. Moreover, using PIM-Tree, we must maintain a sliding window twice the size as is needed for using B$^+$-Tree (considering $r = 1$). Consequently, the total memory space needed for IBWJ using PIM-Tree is nearly twice the amount needed as for using B$^+$-Tree.

Scalability – The objectives of this experiment are to first study the overhead of the CC mechanisms and to then examine the scalability of our join algorithm using multiple threads. Figure 12a compares the resulting throughputs corresponding to self-join and two-way join using PIM-Tree under a varying number of threads against the single-threaded implementation without CC. The results show that enforcing CC causes performance degradation of nearly 40% and 26% for two-way join and self-join, respectively, mainly as a result of the locking overhead. As we increase the number of

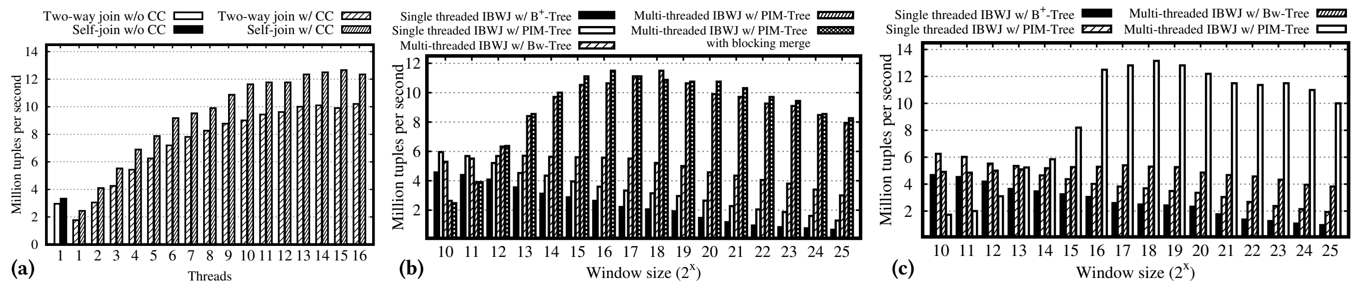

Figure 12: a) Comparison of parallel IBWJ using PIM-Tree utilizing varying number of threads against the single-threaded implementation without concurrency control (CC) ($w = 2^{20}$). b) Throughput comparison of single-threaded and multithreaded two-way join. c) Performance comparison of single-threaded and multithreaded index-based self-join.

threads from one to eight, the performances of two-way join and self-join increase to 4.6 and 4 times the single-threaded implementation with CC, respectively. Moreover, the results reveal that enabling hyperthreading (16 threads) increases the throughput by 24%, and the mentioned improvements increase to 5.7 and 5. As the number of concurrent tasks in the system increases, there is a higher probability of congestion between working threads to access shared resources, which results in longer waiting times and prevents a perfectly linear scale up.

Multithreading efficiency – In this experiment, we study the efficiencies of our multithreading approach and nonblocking merge, and we also compare PIM-Tree to the state-of-the-art parallel indexing tree, Bw-tree. Figure 12b shows the throughput performance of five different implementations of the two-way IBWJ: (1) single-threaded IBWJ using B^+-Tree, (2) single-threaded IBWJ using PIM-Tree, (3) parallel IBWJ using Bw-tree, (4) parallel IBWJ using PIM-Tree, and (5) parallel IBWJ using PIM-Tree with blocking merge.

The results of parallel IBWJ using PIM-Tree show that using blocking and nonblocking merge techniques results in similar performances, while blocking merge is slightly faster than nonblocking merge because of the less complicated mechanism it uses to perform blocking merge operations. Moreover, the results reveal that our parallel approach is effective for window sizes larger than 2^{14}. For the smaller evaluated window sizes (2^{10} to 2^{13}), merge operations occur very often, which leads to frequent linear window scans during merge operations, and thus system performance declines. For window sizes between 2^{15} and 2^{25}, our parallel IBWJ using PIM-Tree yields on average 7.5 and 3.7 times higher throughput than the single-threaded IBWJ using B^+-Tree and PIM-Tree, respectively. The greatest improvement is achieved for the largest evaluated window size (2^{25}), which resulted in improvement increases of 12 and 5.3 times. The evaluation results of IBWJ using Bw-tree reveal that Bw-tree is also not effective for the smaller evaluated window sizes (2^{10} to 2^{13}) because of the high conflict between threads during index operations. For window sizes between 2^{14} and 2^{25}, parallel IBWJ using Bw-tree results in 1.8 times higher throughput than our single-threaded IBWJ using PIM-Tree, on average. For the same range of window sizes, our parallel IBWJ using PIM-Tree outperforms the Bw-tree-based implementation by a factor of 2.2 on average. Although our PIM-Tree achieves better performance than Bw-tree, we do not seek to challenge Bw-Tree in this work since Bw-tree is designed as a generic parallel indexing tree that is highly efficient for OLTP systems where the majority of queries are read accesses (more than 80% [19]), whereas our design is specifically tuned for highly dynamic systems such as data stream indexing with a significantly higher rate of data modification.

Figure 12c presents the performance comparison of the parallel and single-threaded IBWJ implementations for self-join. Similar to the experiment on two-way window joins, parallel self-join using PIM-Tree is not effective for the smaller evaluated window sizes (2^{10} to 2^{15}). For window sizes between 2^{16} to 2^{25}, parallel self-join using PIM-Tree achieves 7 and 4 times higher throughput than the single-threaded self-join using B^+-Tree and PIM-Tree, respectively.

Impact of skewed data – We now study the impact of the tuple value distribution on the performance of parallel IBWJ using PIM-Tree in two experiments. First, we examine IBWJ using three differently skewed distributions, including a Gaussian distribution ($\mu = 0.5, \sigma = 0.125$) and two differently parameterized Gamma distributions (k = 3, θ = 3 and k = 1, θ = 5), and we compare them with the result of using a uniform distribution. For each evaluation, we adjust the band join predicate to keep the average match rate equal to two. Figure 13a presents the evaluation results ($w = 2^{20}$). The uniform distribution of the join attributes always results in the highest throughput, although the differences are not significant. On average, the resulting throughput of IBWJ using PIM-Tree for uniformly distributed join attributes is between 2% and 4% higher than for Gaussian and Gamma distributions, respectively.

In the second experiment, we examine the impact of a dynamic tuple value distribution on the performance of IBWJ using PIM-Tree. In contrast to the previous experiment where the distribution of values was fixed, we now

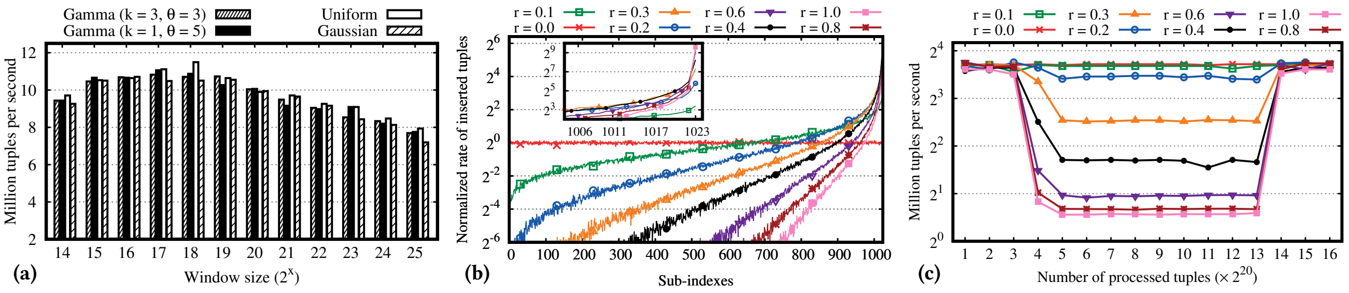

Figure 13: a) Evaluation of parallel IBWJ using PIM-Tree for different tuple value distributions. b) Distribution of inserts among subindexes during drifting Gaussian distributions. c) Evaluation of multithreaded index-based self-join using PIM-Tree for shifting Gaussian distributions.

study PIM-Tree performance under a dynamic value distribution, which results in a skewed distribution of inserts among subindexes. For this purpose, we create a tuple sequence in which tuple values are generated based on a shifting Gaussian distribution, and we then evaluate the performance of parallel index-based self-join using PIM-Tree with this tuple sequence ($w = 2^{20}$). The tuple sequence consists of three phases. In the first phase, the tuples are generated according to the fixed Gaussian distribution $\mathcal{N}(0.5, 0.125)$ ($\mu = 0.5, \sigma^2 = 0.125$). During the middle phase, the distribution of tuple values is linearly shifting from $\mathcal{N}(0.5, 0.125)$ to $\mathcal{N}(r + 0.5, 0.125)$, where the constant value r defines the speed of the distribution change; thus, the larger r is, the faster the mean value of the Gaussian distribution shifts. In the last phase, the tuples are generated according to the Gaussian distribution $\mathcal{N}(r + 0.5, 0.125)$. We set the lengths of these three phases to 3M (3×2^{20}), 10M and 3M tuples, respectively. D_I is set to 4, which results in 1024 subindexes considering $f_{ib} = 32$ and $w = 2^{20}$. Figure 13b illustrates the normalized distribution of insert operations among T_I's subindexes during distribution shifts (second phase) for different values of r ranging from 0 to 1. It follows that inserts are spread among subindexes equally when the tuple value distribution is fixed ($r = 0$), and as r increases, the distribution of inserts becomes more skewed. For the highest value of r ($r = 1$), the insert distribution is highly skewed such that 77% of all inserts are assigned to a single subindex, and there are almost no inserts assigned to the other 70% of subindexes. Figure 13c presents the evaluation results for multiple values of r ranging from 0 to 1. The join performance during the distribution change depends on how fast the distribution shifts: slow, moderate or fast. During slow distribution shifts ($r = 0.1, 0.2$), there is almost no decrease in the stream join performance, which indicates that PIM-Tree is able to gracefully tolerate slow changes in the tuple value distribution. For moderate distribution shifts ($r = 0.4, 0.6$), the system performance decreases to 35% on average, which is due to high partition locking congestion. The lowest performance results from fast distribution shifts ($r = 0.8, 1.0$), where the performance decreases to 16%. The join performances for $r = 0.8$ and $r = 1.0$ are nearly identical, which indicates that partition locking congestion is close to its peak. Additionally, the results imply that regardless of how fast the distribution shifts during the second phase, as the distribution becomes stationary again in the third phase, partitions in PIM-Tree are adjusted accordingly, and stream join performance recovers.

6 RELATED WORK

Work related to our approach can be classified as follows: Tree indexing, parallel B$^+$-Tree, sliding window indexing, and parallel window join.

Tree indexing – Due to the advances in main memory technology, many databases are currently able to store indexing information in main memory and eliminate the expensive I/O overhead arising from storage to disks. Consequently, a large body of work has explored tree-based in-memory indexing. B$^+$-Tree is a popular modification of B-Tree, which provides better range query performance [3, 10]. T-Tree is a balanced binary tree specifically designed to index data for in-memory databases [21]. Although B-Tree was originally designed as a disk-stored indexing data structure, when properly configured, B-Tree outperforms T-Tree while enforcing CC [25]. Rao et al. [32] extended CSS-Tree [31] to the cache-sensitive B$^+$-Tree (CSB$^+$-Trees), which supports update operations, although B$^+$-Tree outperforms CSB$^+$-Tree in applications that require incremental updates. LSM-Tree is a multilevel data structure that stores each component on a different storage medium [29]. LSM-Tree improves system performance in write-intensive applications using delta merging; however, it does not provide a solution for multi-threaded indexing. Adaptive radix tree (ART) is a high-speed in-memory indexing data structure that exhibits a better memory footprint than a conventional radix tree and better point query performance than B$^+$-Tree [22]. However, B$^+$-Tree outperforms ART in executing range queries [1]. We use B$^+$-Tree as the baseline to evaluate our PIM-Tree since it supports incremental updates and range queries better than other approaches.

Parallel B^+-Tree – Bayer and Schkolnick [4] proposed a CC method for supporting concurrent access in B-Trees based on *coupled latching*, in which threads are required to obtain the associated latch for each index node in every tree traversal. B-link is a B^+-Tree with a relaxed structure that requires fewer latch acquisitions to handle concurrent operations [20]. However, CC methods based on coupled latching are known to suffer from high latching overhead and poor scalability for in-memory systems [7].

PALM is a parallel latch-free B^+-Tree based on bulk synchronous processing [35]. Although this approach is scalable and handles data distribution changes, it requires processing queries in large groups (the authors suggest groups of 8,000 queries to achieve a reasonable scale up). Pandis et al. [30] proposed physiological partitioning (PLP) of indexing data structures on the basis of a multirooted B^+-Tree. Using PLP, the index structure is partitioned into disjoint intervals, and each interval is assigned exclusively to a single thread.

Rastogi et al. [33] introduced a multiversion CC and recovery method in which update transactions create a new version of a node to avoid conflicting with lookup transactions rather than using locks. Optimistic latch-free index traversal is based on node versioning to ensure data consistency during tree traversal, but it does not require the creation of a new physical node to avoid conflicts [7]. However, this approach does not provide an efficient node-merging algorithm, which is critical for preserving an efficient tree structure when the data distribution of tuples in the sliding window changes. Bw-Tree is another optimistic latch-free parallel indexing data structure that utilizes atomic compare and swap (CAS) operations to avoid race conditions [23]. Bw-Tree is designed to simultaneously exploit the computational power of multicore processors and the memory bandwidth of underlying storage, such as flash memories. Among the aforementioned approaches, Bw-Tree is the best choice for use cases with frequent incremental updates, which is why we use it as the baseline for our multithreaded indexing approach.

Sliding window indexing – Golab et al. [15] evaluated different sliding window indexing approaches, such as hash-based and tree-based indexing, for different types of stream operators. Kang et al. [17] evaluated the performance of an asymmetric sliding stream join using different algorithms, such as nested-loop join, hash-based join, and index-based join. Lin et al. [24] and Ya-xin et al. [43] proposed the *chained index* to accelerate index-based stream joins utilizing coarse-grained tuple disposal. However, all of these approaches considered only single-threaded sliding window indexing, thus avoiding concurrency issues resulting from parallel update processing, which is central to the focus of our work.

Parallel window join – Window join processing has received considerable attention in recent years due to its computational complexity and importance in various data management applications. Cell join is a parallel stream join operator designed to exploit the computing power of the cell processor [13]. Handshake join is a scalable stream join that propagates stream tuples along a linear chain of cores in opposing directions [41]. Roy et al. [34] enhanced the handshake join by proposing a fast-forward tuple propagation to attain lower latency. SplitJoin is based on a top-down data flow model that splits the join operation into independent store and process steps to reduce the dependency among processing units [28]. Lin et al. [24] proposed a real-time and scalable join model for a computing cluster by organizing processing units into a bipartite graph to reduce memory requirements and the dependency among processing units.

All these approaches are based on context-insensitive window partitioning. Although these methods are effective for using nested loop join or for memory-bounded joins with high selectivity, context-insensitive window partitioning causes redundant index operations using IBWJ, which limits the system efficiency.

7 CONCLUSIONS

In this paper, we presented a novel indexing structure called PIM-Tree to address the challenges of concurrent sliding window indexing. Stream join using PIM-Tree outperforms the well-known indexing data structure B^+-Tree by a margin of 120%. Moreover, we introduced a concurrent stream join approach based on PIM-Tree, which is, to the best of our knowledge, one of the first parallel index-based stream join algorithms. Our concurrent solution improved the performance of IBWJ up to 5.5 times when using an octa-core (16 threads) processor.

The directions for our future work are twofold: (1) developing a distributed stream band join and (2) extending PIM-Tree to support the indexing of multidimensional data. In this paper, we focused on parallelism within a uniform shared memory architecture. A further challenge, but an altogether different problem, is to develop a parallel IBWJ algorithm for nonuniform memory access (NUMA) architectures, which requires addressing two main concerns. First, a range partitioning technique that distributes a workload uniformly among operating cores is needed. Second, a repartitioning scheme that alleviates the overhead of data transfer between memory nodes in a NUMA system is needed. Moreover, with respect to supporting multidimensional data, PIM-Tree is designed to index one-dimensional data. Multidimensional indexing is a vital requirement for many applications, specifically those that utilize spatiotemporal datasets. Thus, a further direction is the design of a multidimensional PIM-Tree.

8 ACKNOWLEDGMENTS

This research has been supported by the Alexander von Humboldt Foundation.

REFERENCES

[1] V. Alvarez, S. Richter, Xiao Chen, and J. Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *ICDE*. 1227–1238.

[2] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *ACM Sigmod Record* (2001), 109–120.

[3] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *SIGFIDET*. 107–141.

[4] R. Bayer and M. Schkolnick. 1977. Concurrency of operations on B-trees. *Acta Informatica* (1977), 1–21.

[5] Timo Bingmann. 2008. STX B+tree C++ template classes. *URL http://panthema. net/2007/stx-btree* (2008).

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, et al. 2015. Apache flink : Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015).

[7] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, et al. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. *VLDB* (2001), 181–190.

[8] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, et al. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *VLDB* (2014), 401–412.

[9] Daniele Dell' Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. 2017. Stream reasoning: A survey and outlook : A summary of ten years of research and a vision for the next decade. *Data Science* (2017), 59–83.

[10] Ramez Elmasri. 2008. *Fundamentals of database systems*. Pearson Education India.

[11] Xiaoming Gao, Emilio Ferrara, and Judy Qiu. 2015. Parallel clustering of high-dimensional social media data streams. In *CCGrid*. 323–332.

[12] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. 2008. SPADE: the system s declarative stream processing engine. In *SIGMOD*. 1123–1134.

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

[14] Pawel Gepner and Michal Filip Kowalik. 2006. Multi-core processors: New way to achieve high system performance. In *PARELEC*. 9–13.

[15] Lukasz Golab, Shaveen Garg, and M Tamer Özsu. 2004. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology*. 712–729.

[16] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, et al. 2013. IBM streams processing language: analyzing big data in motion. *IBM Journal of Research and Development* (2013), 7–1.

[17] Jaewoo Kang, Jeffery F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Data Engineering, International Conference on*. 341–352.

[18] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, et al. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*. 555–569.

[19] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, et al. 2011. Fast Updates on Read-optimized Databases Using Multi-core CPUs. *VLDB* (2011), 61–72.

[20] Philip L Lehman et al. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems* (1981), 650–670.

[21] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*. 294–303.

[22] Viktor Leis, Alfons Kemper, and Tobias Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.

[23] Justin J Levandoski, David B Lomet, and Sabyasachi Sengupta. 2013. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[24] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *SIGMOD*. 811–825.

[25] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. 2000. T-tree or B-tree: Main memory database index structure revisited. In *ADC*. 65–73.

[26] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, et al. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *USENIX ATC 17*. 617–629.

[27] Giovanni Montana, Kostas Triantafyllopoulos, and Theodoros Tsagaris. 2008. Data stream mining for market-neutral algorithmic trading. In *Proceedings of the 2008 ACM symposium on Applied computing*. 966–970.

[28] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. In *USENIX Annual Technical Conference*. 493–505.

[29] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* (1996), 351–385.

[30] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: Page Latch-free Shared-everything OLTP. *VLDB* (2011), 610–621.

[31] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*. 78–89.

[32] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *SIGMOD*. 475–486.

[33] Rajeev Rastogi, S. Seshadri, Philip Bohannon, et al. 1997. Logical and Physical Versioning in Main Memory Databases. In *VLDB*. 86–95.

[34] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *VLDB* (2014), 709–720.

[35] Jason Sewall, Jatin Chhugani, Changkyu Kim, et al. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *VLDB* (2011), 795–806.

[36] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-Tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*. 1523–1538.

[37] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2019. Parallel Index-based Stream Join on a Multicore CPU. https://arxiv.org/pdf/1903.00452.pdf. (2019). arXiv:cs.DB/1903.00452

[38] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. 417–432.

[39] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *VLDB* (2020), 616âĂŞ628.

[40] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD* (2005), 42–47.

[41] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Sigmod*. 625–636.

[42] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, et al. 2014. Storm@Twitter. In *SIGMOD*. 147–156.

[43] Yu Ya-xin, Yang Xing-hua, Yu Ge, and Wu Shan-shan. 2006. An indexed non-equijoin algorithm based on sliding windows over data streams. (2006), 294–298.

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Communication of the ACM* (2016), 56–65.

[45] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, et al. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *VLDB* (2019), 516–530.

[46] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, et al. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2015), 1920–1948.

[47] Linfeng Zhang and Yong Guan. 2008. Detecting click fraud in pay-per-click streams of online advertising networks. *ICDCS*, 77–84.