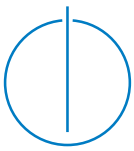
Seyedamirhesam Shahvarani

Parallel In-memory Data Processing using Modern Hardware

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Parallel In-memory Data Processing using Modern Hardware

Seyedamirhesam Shahvarani

Vollständiger Abdruck der von der Fakultät für Informatik der Technische Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: apl. Prof. Dr. Georg Groh

Prüfer der Dissertation:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. Tilmann Rabl

Die Dissertation wurde am 07.04.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.09.2021 angenommen.

Abstract

Nowadays, information systems face an unprecedented challenge of handling astonishing amounts of data at an ever-increasing pace. The performance of conventional disk-based data management systems is no match for this challenge. Therefore, in-memory data management and analytics become an interesting alternative for many applications. The growing memory capacity of computing servers made it feasible for many applications to use main memory as the main storage medium. By eliminating the expensive overhead of disk I/O and utilizing the greater performance of main memory, in-memory data management systems offer support for low latency services and real-time analytics. However, data management systems are facing difficulties and challenges to exploit the superior bandwidth of main memory. Simply replacing disk-based storage with main memory does not address this issue. Because of the different characteristics of main-memory, the conventional approaches used in disk-centric systems are not efficient for in-memory settings. The emergence of new compute platforms further compounds this challenge. To overcome the limitations of serial execution, chip manufactures have shifted toward various forms of parallel computing, such as multi-core processors, multi-socket servers and many-core accelerators. A single-threaded approach is no longer capable of exploiting compute resources, and parallel execution is necessary for any modern high-performance application. Therefore, dedicated and mindful solutions are desired for optimal utilization of memory bandwidth and computational capabilities of modern hardware.

This thesis explores new approaches for parallel in-memory data indexing for various applications and computing platforms. Firstly, we propose a unique design for B$^+$-Tree based on a heterogeneous CPU-GPU computing model. Utilizing a hybrid memory layout, HB$^+$-Tree exploits the compute power of both CPU and GPU platforms simultaneously in order to accelerate lookup-intensive operations. Secondly, we propose a novel data structure, PIM-Tree, to address the challenges of indexing data streams. Combining two complementary techniques, range partitioning and delta update, PIM-Tree enables high-performance range queries over highly dynamic data. Lastly, we propose a distributed solution for parallel *k*NN join for streaming data. Our solution is based on a multi-stage *k*NN execution process that is capable of utilizing a multi-socket compute platform. Combined with an adaptive data repartitioning technique, our approach offers a scalable and real-time solution for the problem of knn join in streaming settings.

Zusammenfassung

Heutzutage stehen Informationssysteme vor der nie dagewesenen Herausforderung, erstaunliche Datenmengen in einem immer höheren Tempo zu verarbeiten. Die Leistung herkömmlicher festplattenbasierter Datenverwaltungssysteme ist dieser Herausforderung nicht gewachsen. Daher wird die In-Memory-Datenverwaltung und -Analyse für viele Anwendungen zu einer interessanten Alternative. Die wachsende Hauptspeicherkapazität von Computerservern machte es für viele Anwendungen möglich, den Hauptspeicher als Hauptspeichermedium zu nutzen. Durch die Eliminierung des teuren Overheads von Platten-E/A und die Nutzung der höheren Leistung des Hauptspeichers, bieten In-Memory-Datenverwaltungssysteme Unterstützung für Dienste mit niedriger Latenz und Echtzeit-Analyse. Die überlegene Bandbreite des Hauptspeichers zu nutzen, stellt Datenverwaltungssysteme jedoch vor Schwierigkeiten und Herausforderungen. Das einfache Ersetzen von festplattenbasiertem Speicher durch Hauptspeicher löst dieses Problem nicht. Aufgrund der unterschiedlichen Eigenschaften des Hauptspeichers sind die konventionellen Ansätze, die in festplattenzentrierten Systemen verwendet werden, für In-Memory-Umgebungen nicht effizient. Diese Herausforderung wird durch das Aufkommen neuer Rechnerplattformen noch verschärft. Um die Einschränkungen der seriellen Ausführung zu überwinden, haben sich die Chiphersteller zu verschiedenen Formen des parallelen Rechnens, wie Mehrkernprozessoren, Multisockel-Server und Mehrkernbeschleuniger verlagert. Ein Single-Threaded-Ansatz ist nicht mehr in der Lage, die Rechenressourcen auszunutzen, und die parallele Ausführung ist eine Notwendigkeit für jede moderne Hochleistungsanwendung. Daher sind dedizierte und achtsame Lösungen für eine optimale Ausnutzung der Speicherbandbreite und der Rechenkapazitäten moderner Hardware erwünscht.

In dieser Arbeit werden neue Ansätze für die parallele Indizierung von In-Memory-Daten für verschiedene Anwendungen und Rechnerplattformen untersucht. Zunächst schlagen wir ein einzigartiges Design für B^+-Tree vor, das auf einem heterogenen CPU-GPU-Rechenmodell basiert. Durch die Verwendung eines hybriden Speicher-Layouts nutzt HB^+-Tree die Rechenleistung von CPU- und GPU-Plattformen gleichzeitig aus, um lookup-intensive Operationen zu beschleunigen. Zweitens schlagen wir eine neuartige Datenstruktur vor, PIM-Tree, um die Herausforderungen der Indizierung von Datenströmen zu bewältigen. Durch die Kombination von zwei komplementären Techniken, der Bereichspartitionierung und Delta-Updates, ermöglicht PIM-Tree hochperformante Bereichsabfra-

gen über hochdynamische Daten. Schließlich schlagen wir eine verteilte Lösung für einen parallelen *k*NN-Join für Datenströme vor. Unsere Lösung basiert auf einem mehrstufigen *k*NN-Ausführungsprozess, der in der Lage ist, eine Multi-Socket-Computerplattform zu nutzen. Kombiniert mit einer adaptiven Daten-Repartitionierungstechnik bietet unser Ansatz eine skalierbare Echtzeitlösung für das Problem des knn-joins in Streaming-Einstellungen.

To the memory of my friend

Saamer Akhshabi

(1987-2014)

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Dr. Hans-Arno Jacobsen for encouraging me to aim high in my academic career and for supporting my PhD studies with sharing his invaluable experience and providing my scholarship. I also would like to thank Prof. Dr. Tilmann Rabl as the external examiner and Prof. Dr. Georg Groh for accepting to chair the examination committee.

To my colleagues at university, I would like to thank you all for your thoughtful comments, technical discussions, and also for all the remarkable and joyful moments at the chair.

Last but not least, I would like to express special thanks to my parents, my brother Amin, and to my friends. This journey wouldn't be possible without your support and encouragement.

Contents

1

Introduction

In recent decades, in-memory data management systems have become increasingly popular. DRAM prices have declined with advances in memory technology, and DRAM has become viable for many applications to store their business data entirely in main memory instead of in disk-based storage. By eliminating the expensive I/O operations, in-memory data management systems offer significantly better performance than disk-stored databases. However, effective resource utilization remains a challenge for in-memory systems. Conventional data structures and data management algorithms, which are optimized according to the characteristics of disk-based storage, are not capable of harnessing the superior bandwidth of in-memory systems. This shortcoming initiated an effort in the database management community to rethink the algorithms and data structures to exploit modern hardware.

This thesis addresses some challenging problems in in-memory data processing using modern hardware. Because of the unique complexity of every computing platform, a single solution often cannot result in an optimal performance when using different underlying hardware. Therefore, a dedicated solution is needed to exploit the computational power of each computing platform. In particular, this work studies the following three subjects: accelerated B^+-Tree utilizing a CPU-GPU heterogeneous platform, parallel join operation over data streams utilizing a multicore processor, and distributed *k*NN join operation on data streams.

1.1 Motivation

The landscape of data management systems has been rapidly evolving in the last decade. Information systems gather astonishing amounts of data from various sources at an ever-increasing pace, and new classes of applications and services have emerged with advances in communication technologies and mobile devices [1, 2]. Currently, more than 4.6 billion people are considered internet users globally, while more than 90% of them have access to mobile devices [3, 4]. The volume of data generated worldwide in the year 2020 is projected to be approximately 30 times higher than that in the year 2010 [5]. As a result of these shifts in technology and applications, data management systems are facing unprecedented challenges. Conventional disk-centric approaches are not effective in confronting these challenges, and in-memory data processing has become the mainstream approach in many data management systems.

For decades, databases relied on hard drives as the main storage, and they utilized main memory for data caching. This trend has come to an end with advances in memory

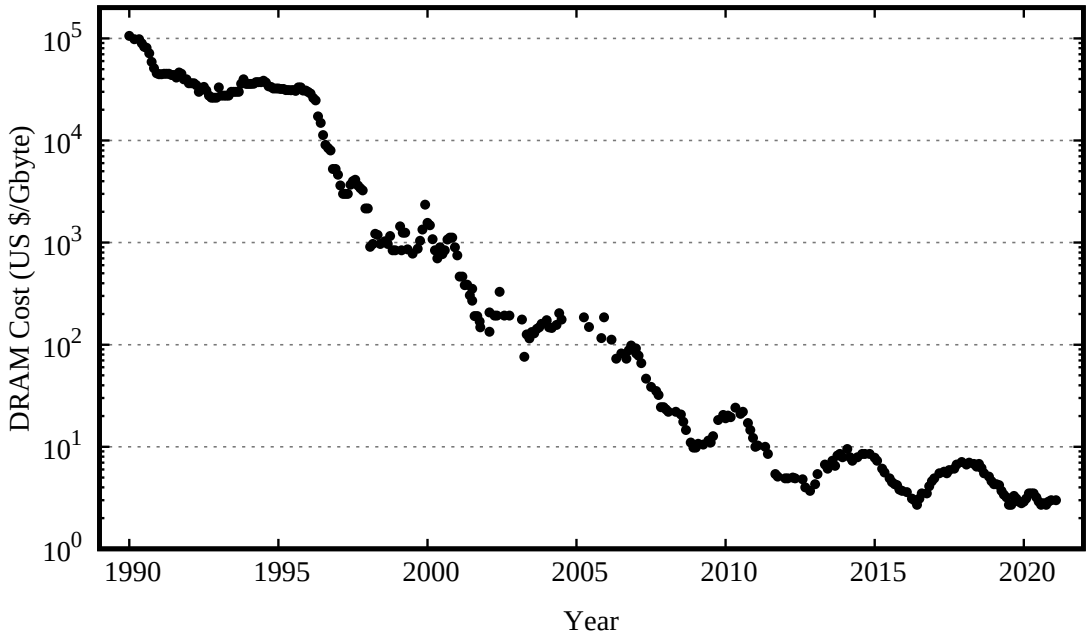


Figure 1.1.1: Cost of dynamic random-access memory (RAM) over time. Since 1990, DRAM prices have decreased by approximately 40 percent a year on average [6].

technology and the growth of the main memory capacity [8]. As illustrated in Figure 1.1.1, the average price for a Gbyte of DRAM decreased to 0.01% in the year 2019 compared to that in the year 1990. As a consequence, the main memory capacity of modern servers increased to a level at which the business information of many enterprise applications could entirely fit in main memory. This enables data management systems to employ main memory as primary storage and rely on conventional disk-based technologies for persistent storage and data backup. By eliminating the expensive I/O overhead, in-memory databases offer significantly better performance than conventional databases and provide capabilities for real-time data analytics and services. Consequently, in-memory databases have become an attractive choice for many applications. However, in-memory systems require different algorithms than those used in disk-centric databases. Algorithms in conventional databases are optimized according to the particular characteristics of disk storage, which are completely different from those of random access memory.

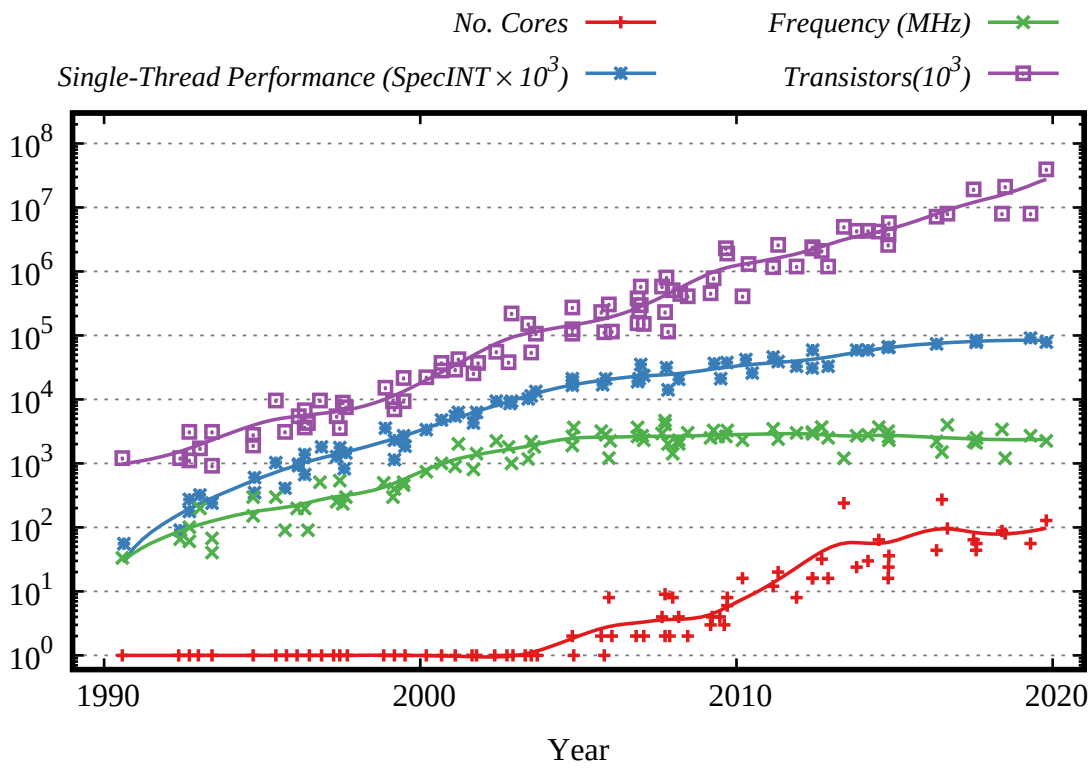


Figure 1.1.2: Evolution of the processor architecture over the years. Despite the increasing number of transistors, the single-thread performance of processors has distinctly decelerated since the year 2003 because of the technical limitations in increasing the processor frequency [7].

Concurrent with the shift toward in-memory data management systems, the processor architecture has undergone a significant evolution in recent decades [9]. Figure 1.1.2 presents the trends in the processor architecture regarding the core number, single-core performance, clock frequency and number of transistors. For a long time, single-core processors were the mainstream architecture, and processor manufacturers relied on instruction-level parallelism and increasing the clock rate to improve the processor performance. However, this trend came to an end because of technological barriers such as power consumption and heat dissipation issues. To confront this problem, processor manufacturers shifted toward multicore processors such as multicore CPUs and many-core GPUs. As a consequence of this change in processor architecture, single-threaded applications are no longer capable of harnessing the computing power of modern servers, and parallel computing has become essential for high-performance applications [10].

1.2 Problem Statement

This thesis addresses three challenging high-performance in-memory data processing problems. First, we address the limited capacity of GPU accelerators for indexing large datasets. Modern GPU accelerators offer a significant amount of computing power, which can be utilized to speed up lookups in tree-based data structures, such as B^+-Tree. However, the amount of available memory in GPUs is more limited compared to that in CPUs; this restricts the usage of GPU accelerators in applications using large datasets. Second, we address the concurrency overhead of indexing highly dynamic data. A parallel indexing solution is needed to unlock the computing power of a multicore processor in computing intensive tasks, such as join operation, in data management systems. Proposing a concurrency control mechanism that enables concurrent update and search operations with minimal overhead is a challenging endeavor. This becomes more prominent in streaming databases where highly dynamic data result in an excessive concurrency overhead. Last, we address the problem of real-time spatial partitioning for *k*NN queries over data streams. Data partitioning is essential for spatial queries in a system with nonuniform memory accesses. A challenge for effective data partitioning is the need to adapt to changes in data distribution. This challenge becomes more complicated in real-time applications where data preprocessing is not applicable.

1.2.1 Limited Capacity of GPU Accelerators for Indexing Large Datasets

B^+-Tree is a dynamic data structure commonly used in data management systems as an indexing data structure [11, 12]. B^+-Tree is based on B-Tree, a self-balancing data structure optimized for sequential data access. In addition to the characteristics inherited from B-Tree, such as balanced height and logarithmic cost of search and update, B^+-Tree performs faster range queries. Although B^+-Tree was originally designed for disk-based storage, it also performs well for in-memory systems if correct configurations are used. Because of its importance in data management systems, multiple efforts have been made to employ the computing power of modern hardware to accelerate data indexing using B^+-Tree [13, 14].

Rapid advances in circuit design and the processor architecture resulted in a remarkable gap between the processor and main memory performances. This issue, which is often referred to as the *memory wall*, becomes a limiting factor in many data-intensive applications [15]. Modern multipurpose processors are highly dependent on data caching to mitigate the memory wall problem. For applications with predictive data access, data caching is a highly effective technique. However, for applications that involve unproductive pointer chasing, such as tree traversal, data caching is not helpful. Therefore, we observe a significant performance drop in CPU-optimized tree traversal for datasets larger than the CPU last-level cache.

GPU platforms offer higher memory bandwidth than CPUs, and previous efforts in realizing tree-based indexing using GPUs demonstrate great potential [16]. In contrast to multipurpose processors, which rely on caching to mitigate the memory latency issue, GPUs are based on massive parallelism and low overhead context switch [17]. These techniques are resilient to data size, and therefore, the tree traversal performance using a GPU is not affected by the size of the tree. However, the amount of memory available to GPUs is relatively small compared to the system main memory. This limits the applicability of GPU-accelerated B^+-Trees.

1.2.2 Concurrency Overhead of Indexing Highly Dynamic Data

The source of information in many applications, such as algorithmic trading, fraud detection and social networking, is a transient and real-time sequence of tuples, known as data streams [18, 19, 20, 21]. Because of the limited capacity of main memory and CPU computational power, it is not feasible to store and process an infinite sequence of data. To address this issue, stream processing systems often require limiting the scope of data streams using a sliding window, which is defined as a fixed number of tuples (count based) or a duration of time (time based).

As in conventional databases, indexing is essential to improve the performance of computationally intensive operators in streaming databases [22]. Window join is a fundamental operator in real-time data analytics that correlates the information of two separate sources. Combining the characteristics of the join operator and dynamicity of streaming data, window join is a computationally demanding operator that can greatly benefit from accelerated window lookups. However, the conventional indexing solutions are not directly applicable in streaming settings, and dedicated approaches are needed.

In an index-accelerated window join, the content of each sliding window is indexed into two separate indexes. Upon the arrival of a new tuple, the window join operator utilizes the index of the opposite sliding window to find matching tuples. The content of a sliding window is highly dynamic, and the associated index must be updated frequently. In contrast to data management systems where search query is the most frequently used operation, support for efficient index update is crucial in a streaming setting. In addition, the particular arrival and departure pattern of tuples in a sliding window could be utilized to propose a more effective indexing solution. Therefore, a dedicated approach is needed to tackle the challenges of indexing highly dynamic workloads as in streaming data management systems.

The challenge of handling frequent index updates becomes compounded by the enforcement of a concurrency control mechanism, which is required to enable parallelism and exploit the computational power of a multicore processor. In conventional databases, the index update rate is relatively low compared to the index lookup, and the concurrency control mechanism used in such systems is designed accordingly. For this reason, these

mechanisms result in suboptimal performance for indexing highly dynamic data, such as those in a sliding window.

1.2.3 Real-time Spatial Partitioning for *k*NN Over Data Streams

With advances in mobile devices and communication technologies, location-aware applications and services have become increasingly popular in the past decades [23, 24, 25]. As the number of their users increases, location-aware applications, such as social networking platforms, recommender systems and location-based games, gather and process astonishing amounts of data at an ever increasing pace [26]. High-performance stream processing solutions become essential for these applications in order to meet their performance requirements.

k nearest neighbor (kNN) join is an important and commonly used operator in many location-aware applications [27]. For two given datasets R and S, kNN join associates each entry from dataset R with its k nearest entries from dataset S. In a streaming setting, the datasets R and S are defined as two separate data streams. Stream kNN join is a useful operator in many scenarios, such as locating the taxis nearest to the customer's location, correlating a tweet with the geospatially nearest tweets and finding the nearest photos to a user in a photo-sharing platform. Combining the computational complexity of kNN search and the join operator with the dynamicity of a data stream, stream kNN join is a computationally intensive operator. Therefore, a single-threaded solution for stream kNN join cannot meet the desired performance, and a scalable multithreaded approach that is capable of exploiting the computational power of modern hardware is desirable for this problem.

In terms of underlying hardware, stream processing systems can be divided into two categories, single-node (scale-up optimized) or multinode (scale-out optimized) [28, 29]. Single-node stream processing systems are often based on a nonuniform memory access (NUMA) architecture, and they are focused on algorithms that utilize the resources of a single-node computer. In contrast, scale-out optimized systems are often based on massive data parallelism and a producer-consumer task distribution pattern to exploit the computational resources of a multinode workstation. Over the last decade, the perfor-

mance of NUMA computers has been highly improved regarding both the computation capability and memory bandwidth. Therefore, single-node stream processing solutions focused on scale-up optimization are sufficient for many applications, and they have become an alternative to scale-out optimization approaches.

For a successful distributed *k*NN join, we require an effective spatial partitioning mechanism that must consider both the workload balance and scalability at the same time. Simple data partitioning approaches such as round-robin may provide a uniform load distribution among working operators, but they are not scalable solutions for *k*NN join. When using round-robin partitioning, we must query every individual partition and combine these local *k*NN results to find the global *k*NN result because tuples are distributed among partitions independent of their values. For this reason, the cost of *k*NN queries increases almost linearly with the number of partitions, which limits the system scalability. Approaches based on hash partitioning are not effective for *k*NN join either because hash functions do not preserve the spatial distances between tuples. Utilizing a space partitioning function, we can restrict the scope of *k*NN queries to a limited subset of partitions and maintain the scalability of our system. However, it is a challenging endeavor to propose a space partitioning mechanism that results in a uniform load distribution among partitions and adapts to data distribution changes in real time.

Although there are many data partitioning approaches for multithreaded *k*NN join over prestored datasets, these partitioning approaches are either incompatible with or inefficient for data stream processing. Data preprocessing is a commonly used technique to partition prestored datasets. However, this approach of data partitioning is not applicable to streaming settings. Furthermore, the content of the sliding window is highly dynamic, and data partitioning must be continuously adjusted with data distribution changes in real time.

1.3 Approach

In this section, we provide a description of our approaches that we proposed to address the three problems introduced in the previous section. Each problem that this thesis

addresses utilizes a unique computing device. Therefore, there are unique techniques and design decisions to exploit the computing resources for each problem.

1.3.1 GPU-Accelerated B$^+$-Tree Based on a Hybrid Memory Layout

High-performance data indexing using either a CPU or a GPU is subject to different trade-offs. GPUs are equipped with higher bandwidth memory modules, and parallel data indexing using a GPU outperform that using CPU platforms. However, the capacity of the system main memory is often larger than the amount of memory available to a GPU. Therefore, it is not feasible to employ a GPU for indexing a large dataset, and high-performance data indexing using a CPU is the only viable option for these scenarios. To address this issue, we propose a hybrid layout for B$^+$-Tree (HB$^+$-Tree) that is able to leverage both CPU and GPU memories at the same time. HB$^+$-Tree employs the computing power of a GPU accelerator for searching on a dataset larger than the GPU dedicated memory. Our main objective is to combine the higher capacity of the system memory with the superior bandwidth of the GPU platform to achieve high-performance data indexing over a large volume of data. To improve the effective memory bandwidth, we scatter the nodes across CPU and GPU memories in a way that enables simultaneous utilization of both memory modules. Furthermore, we propose a heterogeneous search algorithm that minimizes the costs of communication between the CPU and GPU.

Figure1.3.1 illustrates the overall architecture of HB$^+$-Tree and how it resides in memory. HB$^+$-Tree partitions inner nodes and leaf nodes into two separated segments, namely, the I-segment and L-segment, respectively. A search query in HB$^+$-Tree starts from the root node in the I-segment; after passing all the inner nodes, the query continues in the L-segment to locate the target key-value pair. The L-segment only resides in CPU memory, and leaf nodes are optimized according to the CPU memory characteristics. In contrast, the I-segment resides in both CPU and GPU memories, and inner nodes are configured for optimal search performance using the GPU. To update HB$^+$-Tree, all changes are initially applied to the I-segment residing in CPU memory, and then, the copy in GPU memory is updated accordingly. Our rationale for this hybrid design is to exploit the computing power of the GPU for processing inner nodes, which correspond to a large portion of the tree traversal overhead, and utilize the larger CPU capacity to

store leaf nodes.

To complement our hybrid memory layout, we propose a heterogeneous algorithm for tree traversal utilizing both the CPU and GPU. To reduce the communication overhead between the CPU and GPU, we group queries into batches. Our heterogeneous search algorithm processes a query batch in four steps. First, the query batch is transferred from CPU memory to GPU memory. Second, the GPU traverses all the inner nodes for each query in the batch and generates intermediate results. Third, the intermediate results are transferred to CPU memory. Finally, the CPU resumes the search operation using the intermediate results for each query in the batch. We improve the resource utilization of our heterogeneous algorithm by applying stage pipelining and double buffering techniques. Double buffering enables our solution to execute two query batches at a time, and stage pipelining improves the system performance by overlapping the different stages of two active query batches.

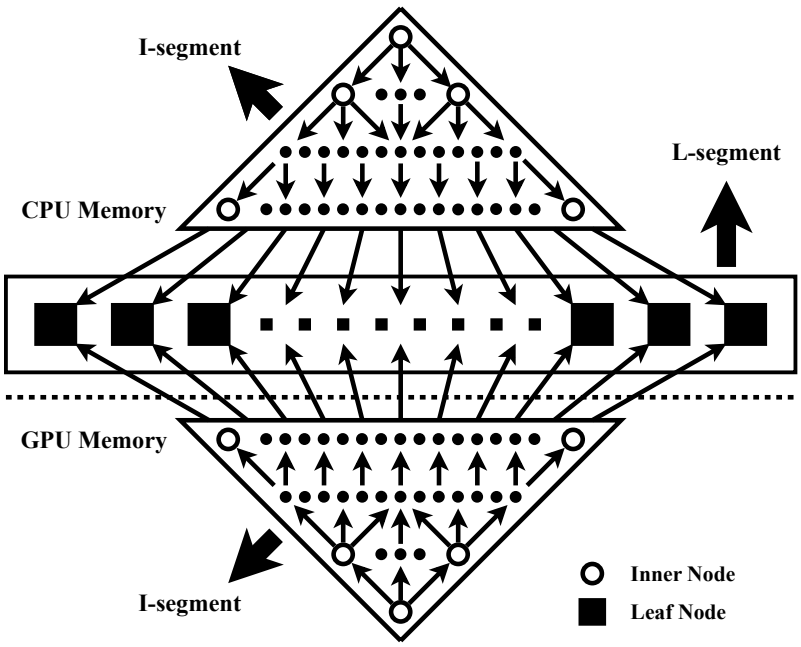


Figure 1.3.1: Distribution of HB^+-Tree nodes among CPU and GPU memories. The leaf nodes reside only in CPU memory, while the leaf nodes are duplicated in both CPU and GPU memories.

1.3.2 Parallel Data Indexing Based on Range Partitioning

We propose a novel indexing data structure, called the partitioned in-memory merge tree (PIM-Tree), which is designed to address the challenges inherent to concurrent data indexing in a highly dynamic setting. The main objective of PIM-Tree is to exploit the computational resources of a multicore processor in the application of multithreaded window join on the basis of uniform memory access. PIM-Tree is a two-stage data structure based on two known techniques, data partitioning and delta updating. The combination of these two techniques enables PIM-Tree to support frequent update queries with a low concurrency overhead, which is highly advantageous for data indexing in streaming environments. In contrast to prior approaches that rely on resolving concurrency at the tree node level, parallelism in PIM-Tree is based on multipartition design, which enables PIM-Tree to benefit from concurrent operations on disjoint ranges of values and relies on the distribution of queries. Therefore, operations in PIM-Tree are as efficient as those in single-threaded B$^+$-Tree, and the only overhead is to acquire a single mutex per operation to avoid race conditions.

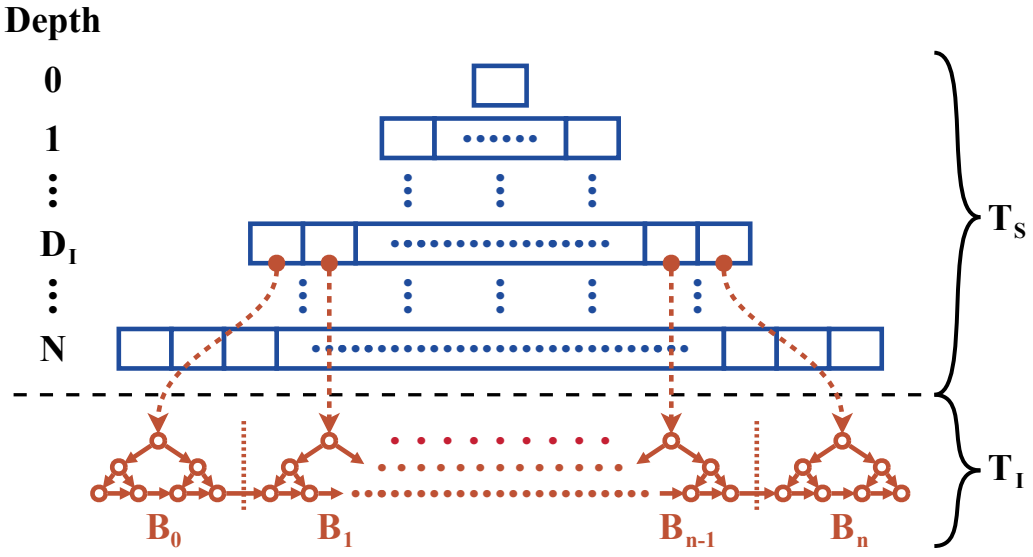


Figure 1.3.2: Overall architecture of PIM-Tree. T_S and T_I are the immutable and mutable components, respectively. T_I is a chain of B$^+$-Trees attached to T_S at insertion depth D_I. T_S is an immutable B$^+$-Tree optimized for search queries.

PIM-Tree consists of two segments, (T_I) and (T_S). T_S is an immutable B$^+$-Tree optimized for search queries and bulk updates, and T_I is a set of n independent insert-efficient B$^+$-Trees (B_i, $1 \leq i \leq n$) that are attached to T_S at a specific depth D_I. To create a single list of all elements in T_I, the tail leaf node of each subindex (B_i, $1 \leq i < n$) is connected to the head leaf node of its successor (B_{i+1}). Each B_i is assigned the same range of values as the i-th inner node of T_S at depth D_I. To query a PIM-Tree, both the T_I and T_S components need to be searched.

All new tuples are initially inserted into T_I. When the size of T_I reaches a predefined threshold, the entire T_I is merged into T_S, and at the same time, the expired tuples in T_S are eliminated. The update routine inserts a tuple in a two-step operation. First, it searches T_S until depth D_I to identify the matching subindex assigned the range that includes the given tuple, and second, the update routine inserts the tuple into the identified subindex using the B$^+$-Tree insert algorithm.

To coordinate concurrent operations in T_I, each subindex is assigned a mutex. All operations, both insert and search, must acquire the associated mutex before performing any operation for a subindex. Furthermore, the last leaf node of each subindex is flagged such that the search routine identifies a move from the subindex to its successor. While a thread is scanning leaf nodes, it may move from a subindex to its successor. In this scenario, the scanning thread must first acquire the successor's mutex and then release the mutex for the current subindex. T_S is a read-only data structure; therefore, there is no need for a concurrency control mechanism to avoid race conditions.

To complement our data structure PIM-Tree, we propose a parallel join algorithm based on adaptive load distribution and a shared work queue. Our join algorithm consists of four steps: task acquisition, result generation, index update, and result propagation. The first step for each thread is to acquire a task, which is a set of input tuples. Then, the thread generates join results by querying the window indexes and stores them in a local buffer. In the third step, the thread updates the window indexes according to the new tuples. Finally, it propagates the join results to the output stream.

1.3.3 Adaptive Data Partitioning Based on Real-time Load Monitoring

We propose adaptive distributed stream *k*NN join (ADS-kNN), a scalable solution for real-time *k*NN join in a streaming environment. In contrast to distributed *k*NN solutions based on data preprocessing, load balancing in ADS-kNN is based on online data analysis and repartitioning. Therefore, ADS-kNN does not require any prior knowledge about the distribution of input data, which makes it suitable for real-time data processing.

Figure [**fig:ads**] illustrates the overall architecture of ADS-kNN, which consists of four types of operators: dispatcher, mapper, join-core and load-balancer. The dispatcher receives all input tuples and distributes them between mapper operators. The mapper operators forward the tuples to their corresponding join-cores based on a partitioning map. The join-cores store tuples in their local memory and produce the *k*NN results. Concurrent with the *k*NN operation, the load-balancer continuously monitors the workload distribution among join-cores and generates a new partitioning map if required. Given *n*

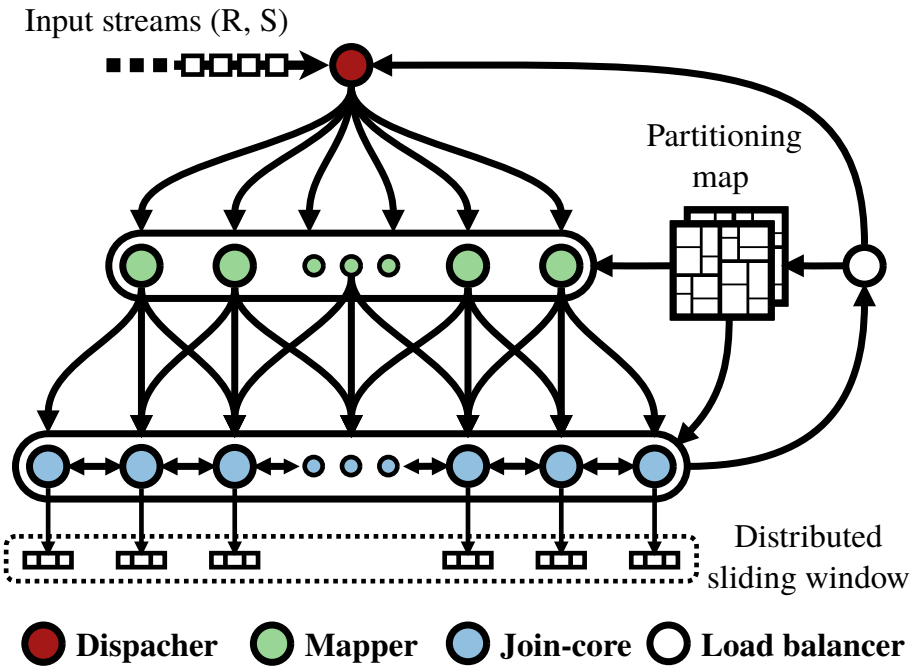


Figure 1.3.3: Overall architecture of ADS-kNN and the communications between its operators.

join-cores, ADS-kNN requires a partitioning map that divides the domain space into n nonoverlapping subspaces. The content of the sliding window is divided into n partitions according to this partitioning map, and each partition is stored in a join-core. The kNN query execution in ADS-kNN is a multistage operation that enables low-latency result delivery and efficient utilization of available computational resources.

The data partitioning in ADS-kNN is based on two separate processes, workload monitoring and data repartitioning. The join-cores periodically generate statistical information about their workload and submit it to the load-balancer. If it detects that the current data partitioning is not effective, then the load-balancer initiates the data repartitioning process. To generate a new partitioning map, the join-cores submit the load approximation of their local data to the load-balancer. Then, the load-balancer generates a new partition map accordingly and sends it to all the mappers and join-cores. We propose two data partitioning replacement methods, lazy and instant. Instant partitioning replacement is a blocking operation in which join-cores instantly redistribute tuples according to the new partitioning map. This approach always maintains the load distribution in an efficient state. In contrast, lazy partitioning replacement is a nonblocking operation, but it takes a longer time for the system performance to recover.

To further optimize the performance of ADS-kNN, we propose a unified operator, mapper-joiner, which operates as both a mapper operator and a join-core. Setting a fixed number of individual mappers and join-cores in the system cannot properly deal with dynamic data distribution changes. To address this issue, we propose the unified mapper-joiner operator, which relies on adjusting the execution time spent on each operation rather than on adapting the number of operators. Using n mapper-joiners is logistically the same as having n mappers and n join-cores in the system. In such a system, the mapper-joiner automatically increases the time spent on the join operation when neighbor querying increases, and likewise, it increases the mapping time if data partitioning becomes more costly.

1.4 Contributions

The main contributions of this work regarding each described problem are listed in the following.

- **Hybrid CPU-GPU B$^+$-Tree**
  i. We propose a novel data structure, HB$^+$-Tree, based on a hybrid memory layout that enables concurrent utilization of both CPU and GPU memories.
  ii. We develop a load balancing scheme that enables optimal resource utilization for various configurations of CPUs and GPUs.
  iii. We develop an analytical model to examine the influence of the techniques and approaches that we used.
  iv. We propose a CPU-optimized B$^+$-Tree as a baseline for our hybrid solution, which outperforms the state-of-the-art tree-based indexing approach.
  v. We conduct an extensive evaluation to study the effect of different optimization techniques and the efficiency of our hybrid approach using various workloads.

- **Parallel Index-based Window Join**
  i. 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 join in both single- and multithreaded settings.
  ii. We develop an analytical model to compare the costs of window join using the indexing approaches studied in this paper to provide better insight into our design decisions.
  iii. We propose a parallel index-based window join (IBWJ) algorithm that addresses the challenges arising from using a shared index in a concurrent manner.
  iv. We conduct an extensive experimental study of IBWJ employing PIM-Tree and provide a detailed quantitative comparison with state-of-the-art approaches.

- **Distributed Stream *k*NN Join**

  i. We propose a scalable multistage kNN query execution to achieve high-performance and low-latency distributed kNN join.

  ii. We propose an adaptive data partitioning method that adjusts the load distribution according to input data streams.

  iii. We design and develop a lightweight stream processing framework to implement ideas presented in this paper.

  iv. We develop an analytical comparison between our approach and the state-of-the-art to provide better insight into our design decisions.

  v. We conduct an extensive experimental study of ADS-kNN and provide a detailed quantitative comparison with state-of-the-art approaches.

1.5 Organization

The rest of this thesis is organized as follows. Chapter 2 presents the methodologies we used to accomplish the ideas presented in this thesis. Chapter 3 summarizes the papers comprising this thesis. Chapter 4 discusses our approaches in the context of their related approaches. Chapter 5 concludes this thesis.

2

Methodology

In this chapter, we briefly describe the background information and design considerations necessary to understand our solutions proposed in this thesis. In Section 2.1, we explain the computing platforms and programming models that we used in this thesis. In Section 2.2, we describe the key elements of the memory architecture and the related design considerations. Finally, in Section 2.3, we explain the performance profiling tools we use to develop the ideas presented in this thesis.

2.1 Programming Models and Computing Architectures

To implement the approaches presented in this thesis, we used three different computing platforms. In this section, we provide a brief introduction to each computer architecture and its programming model.

2.1.1 Compute Unified Device Architecture (CUDA)

CUDA is Nvidia's framework for general purpose computing utilizing GPU accelerators [17]. The CUDA programming model is based on massive parallelism using thousands

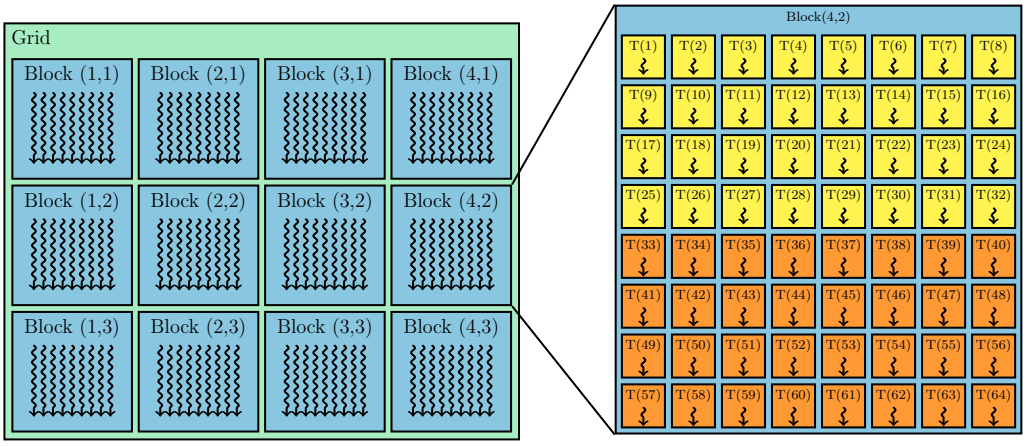


Figure 2.1.1: Arrangement of threads within a grid. In this example, the grid consists of 12 blocks (4 × 3), and each block is composed of 64 threads (8 × 8). Threads within each block are divided into two separate warps, yellow and orange [17].

of threads. Threads are organized into two-level hierarchies consisting of grids and blocks. A grid is at the top of the hierarchy and is a three-dimensional array of blocks, and similarly, each block is a three-dimensional array of threads. Individual blocks and threads are identified using special indexes BlockIdx and ThreadIdx, respectively. All threads in a grid run the same kernel code, which is a function to be executed on a GPU. Figure 2.1.1 depicts the overall arrangement of threads and blocks in a grid.

Effective utilization of GPU resources requires a good understanding of the GPU architecture, including memory components and processing cores. The unit of scheduling in CUDA is a set of 32 threads, which is referred to as a *warp*. All threads in a warp have to execute the same instruction at a time. There is a situation referred to as thread divergence in which threads of the same warp divert into separate code paths, which typically occurs in if-then-else statements. In such scenarios, the entire warp has to execute all individual code paths separately, which results in a significant performance penalty. Thread divergence can be avoided by using a proper thread alignment such that threads that follow the same code path fall in the same warp.

The computing resources in the CUDA platform are organized into a similar hierarchical pattern as its programming model. A GPU consists of several multiprocessors, in which each one is a complex unit composed of various components. These components

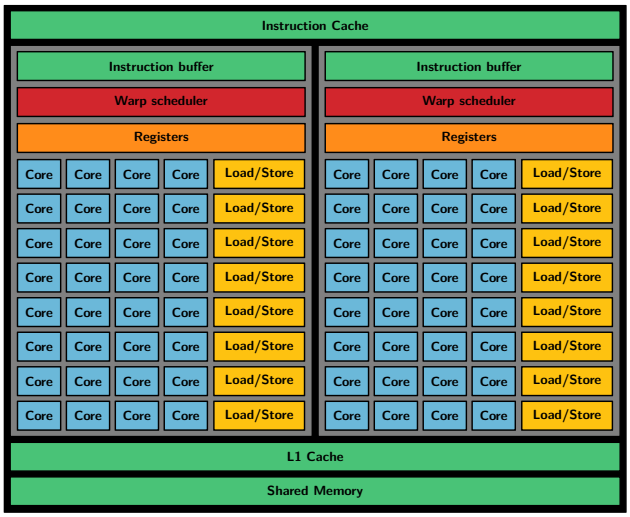


Figure 2.1.2: Overall architecture of a multiprocessor in the CUDA architecture [17].

are illustrated in Figure 2.1.2. To process a grid, each block is assigned to a single multiprocessor, and the threads are scheduled over processing cores. Since CUDA does not support any form of message passing, the only available solution to share information between threads is through read and write on a shared memory module. Therefore, optimal utilization of shared memory resources is of great importance. The CUDA memory architecture consists of six different types of memory modules, global, shared, constant, texture, local and register. Global memory is the largest and slowest module, and it is accessible by all threads. Shared memory is banked memory modules that are accessible by all threads within a single block. Both texture memory and constant memory are read-only modules that are optimized for particular patterns of data accesses. Registers are local to each thread, and they are the fastest type of memory in the CUDA architecture. Whenever there are not enough registers to execute a kernel, the extra needed space is allocated as local memory. The local memory of each thread is only accessible by the thread itself, and the performance is as slow as that of global memory.

2.1.2 Message Passing Interface (MPI)

Single-node computing platforms offer a limited amount of resources regarding both computational power and memory capacity. Distributed computing platforms address

these limitations by utilizing multiple computing nodes connected by a communication network. Processes running on these platforms do not have direct access to each others' memory spaces. Therefore, they need to share data using other mechanisms, such as message passing. The message passing interface (MPI) is a communication standard for parallel programming based on interprocess communication [30]. The MPI program creates multiple processes with individual memory spaces. These processes communicate with each other through message passing, in which data are transferred from the address space of a process to another. The MPI standard only specifies the syntax and semantics of library routines such as names, calling sequence and function outcome, and it is intended to be implemented for different communication networks.

The MPI communicator provides a scope for multiple processes to communicate and synchronize. Among other features, a communicator includes process groups, which specify the participating processes. A group is an ordered set of processes, each assigned a unique rank, which is used as an identifier for communication within the communicator. MPI supports various forms of communication between processes. Regarding the number of participants, MPI operations are divided into two categories, point-to-point and collective. Point-to-point operations, such as sending and receiving a message, occur between exactly two processes, a sender and a receiver. Collective operations are invoked by all processes in a communicator, such as group synchronization, gather and scatter operations. MPI communications are often performed on a two-sided basis, which requires matching send and receive requests from two processes. In contrast, a process may access the memory space of another process using one-sided communication. This communication mode enables better concurrency among processes by decoupling data transmission from process synchronization, and it is better for applications with unpredictable messaging patterns.

2.1.3 Open Multiprocessing (OpenMP)

Parallel computing on a single-node computer has become feasible with the rise of multicore processors. In such systems, processes have access to a shared memory space, which they use to communicate with each other through read and write operations. Although parallel computing in shared memory systems can be performed by utilizing a

generic multithreading library, such as Pthread, these libraries are often not optimized for high-performance computing applications; therefore, there are programming overheads to handle operations such as thread management and communication. In such scenarios, it is more convenient to use a special propose multithreading library, such as OpenMP. Open Multiprocessing (OpenMP) is an application program interface for parallel computing on the basis of a shared-memory architecture [31]. It consists of programming language extension, library routines, and environment variables that are designed to describe parallel execution, thread communication and synchronization. Multithreading in OpenMP is based on the fork-join model, and threads communicate with each other through reads and writes on shared memory. The OpenMP program initiates as a single process, which is referred to as the master thread. The master thread creates a parallel region by forking into multiple threads and ends the region by using the join operation. An OpenMP program might have several parallel regions.

OpenMP supports various forms of parallelism paradigms. Loop parallelism is the most common and simplest type of a parallel region in OpenMP, in which threads collectively process a loop statement. OpenMP offers different thread scheduling patterns, which are specified by compiler directives or environmental variables. Task parallelism is another type of parallel region in OpenMP, in which tasks are dynamically generated into a task queue and executed by operating threads. This form of parallelism is useful for scenarios with unpredictable data access pattern, such as graph processing. Furthermore, it is possible to use OpenMP as a generic multithreading library. In this scenario, the programmer has to explicitly coordinate most aspects of a parallel program, such as thread communication, synchronization and load balancing.

2.2 Memory Optimization

Effective utilization of memory resources is crucial for a successful data-intensive application. Memory hierarchy, including different components such as registers, cache and main memory, is an important element of modern computer hardware, which specifies the data access latency and bandwidth in a system. It is essential for a high-performance application to employ algorithms and approaches that are aware of different aspects of

memory hierarchy to achieve optimal resource utilization [32, 33].

Software pipelining is an optimization technique to improve the loop performance by interleaving instructions of different iterations [34]. It scatters the dependencies between the instructions of a single iteration among multiple loop iterations. As a result, the CPU pipeline can be scheduled in a way that reduces CPU stalls caused by memory latency. Software pipelining is an effective technique to improve the performance of tree data structures. Search operation in tree-based data structures is among the use cases that can be improved by employing software pipelining. Tree traversal involves multiple iterations of node fetch and processing steps. Such scenarios result in frequent stalls in a CPU while the next node in the traversal is loaded into the CPU cache. Software pipelining mitigates this problem by interleaving the node fetch and processing steps of multiple queries such that the CPU time is utilized to process other queries while the next node in another query is loaded into the CPU cache.

Data caching is a widely used technique in modern processors to enhance the performance of accesses to main memory [35]. The CPU cache is a small but high-performance memory module that stores copies of frequently accessed data in main memory in order to provide faster access. The caching mechanism in modern processors is often based on a three-layer structure, L1, L2 and L3. L1 is the most performant with the smallest capacity, while L3 is the layer with the largest capacity and the lowest performance. The unit of data transfer between main memory and the CPU cache is a cache line, which is 64 bytes of data. Optimal utilization of cache line data before they are evicted (called cache line blocking) in essential to improve the effective memory bandwidth. In the context of tree-based data structures, B^+-Trees with nodes equal to a cache line result in a better performance [36]. By exploiting all of the data in a cache line, this configuration results in great cache utilization and reduces the total main memory accesses needed to retrieve data from B^+-Tree.

Paging is a technique used for realizing virtual memory, in which system memory is divided into same-sized blocks called pages [32]. A page table is a data structure that stores translation information between virtual memory pages and physical memory blocks. A part of the page table, the translation lookaside buffer (TLB), is cached by a CPU to improve the translation performance. If the demanded page does not exist in the TLB,

then a page table search is needed to retrieve the translation information. This process is called a page walk, which results in a large performance penalty. Considering the limited capacity of the TLB, the use of a small page size leads to several TLB misses and results in a severe performance penalty. To avoid this situation, there is an opportunity to make use of larger pages called huge pages. However, the capacity of the TLB for storing translation information of huge pages might be limited in a system [37]. Therefore, an application-specific configuration for memory management might be necessary for effective utilization of huge pages.

2.3 Profiling Tools

Program profiling is an analytical technique to gather detailed information about program performance and efficiency, such as time complexity, memory footprint and communication overhead [38, 39]. This information provides better insights into performance bottlenecks and helps developers further optimize their application. There are several approaches for program profiling. Simple techniques such execution time tracking are helpful for detecting the computationally demanding regions of a program. However, these methods do not provide developers with sufficient insights for a better understanding of resource utilization. Thus, utilizing advanced profiling techniques is essential for developing a complex parallel algorithm and studying the impact of various design decisions.

To develop the ideas presented in this thesis, we employed two profiling tools, Nvprof [40] and PAPI [41]. Nvprof is a tool provided by Nvidia to collect and study the profiling data of a GPU-accelerated program. Nvprof provides time analysis of CPU and GPU activities that is beneficial for studying the communication costs between CPU and GPU memories. Furthermore, it provides an analytical study of GPU-accelerated code, which indicates the utilization efficiency of various resources. This information is valuable for detecting performance limiters and optimization opportunities. PAPI (Performance Application Programming Interface) is a portable interface to access low-level performance counters in a CPU. Performance counters are special purpose registers in processors used to record the occurrence of specific events without influencing the processor performance. Utilizing

PAPI, we can track the occurrence of particular events of interest, such as cache misses or TLB misses,

3

Summary of Publications

In this chapter, we provide a brief description of the individual papers comprising this thesis. Overall, this thesis is based on three accepted peer-reviewed publications, provided in Appendixes A, B and C. For each paper, we provide a brief overview of the approach, the key achievements, and a summary of the author's contributions.

3.1 A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms

Reference: Amirhesam Shahvarani and Hans-Arno Jacobsen. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). pages 1523–1538. DOI : https://doi.org/10.1145/2882903.2882918

Full-text version enclosed: Appendix A

Summary:

An in-memory indexing tree is a critical component of many databases. Modern many-core processors, such as GPUs, are offering tremendous amounts of computing power making them an attractive choice for accelerating indexing. How- ever, the memory available to the accelerating co-processor is rather limited and expensive in comparison to the memory available to the CPU. This drawback is a barrier to exploit the computing power of co-processors for arbitrarily large index trees.

In this paper, we propose a novel design for a B^+-Tree based on the heterogeneous computing platform and the hybrid memory architecture found in GPUs. We propose a hybrid CPU-GPU B + -tree, – HB^+-Tree, – which targets high search throughput use cases. Unique to our design is the joint and simultaneous use of computing and memory re- sources of CPU-GPU systems. Our experiments show that our HB + -tree can perform up to 240 million index queries per second, which is 2.4X higher than our CPU-optimized solution.

Author's contributions: Conceived, developed, and implemented the approach. Devised optimizations. Conducted analysis and experimental evaluation. Wrote the paper.

3.2 Parallel Index-based Stream Join on a Multicore CPU

Reference: Amirhesam Shahvarani and Hans-Arno Jacobsen. Parallel Index-based Stream Join on a Multicore CPU. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). pages 2523–2537.
DOI : https://doi.org/10.1145/3318464.3380576

Full-text version enclosed: Appendix B

Summary:

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.

Author's contributions: Conceived, developed, and implemented the approach. Devised optimizations. Conducted analysis and experimental evaluation. Wrote the paper.

3.3 Distributed Stream KNN Join

Reference: Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed Stream KNN Join. In Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21). pages 1597–1609.
DOI : https://doi.org/10.1145/3448016.3457269

Full-text version enclosed: Appendix C

Summary:

*k*NN join over data streams is an important operation for location-aware systems, which correlate events from different sources based on their occurrence locations. Combining the complexity of *k*NN join and the dynamicity of data streams, *k*NN join in streaming environments is a computationally intensive operator, and its performance can be greatly improved by utilizing the computational capabilities of modern non-uniform memory access (NUMA) computing platforms. However, the conventional approaches to *k*NN join for prestored datasets do not work efficiently with the kind of highly dynamic data found in streaming environments.

Therefore, in this paper, we introduce an adaptive scalable stream *k*NN join, named ADS-kNN, to address the challenges that arise when performing the join operation on highly dynamic data, which is common in streaming environments. We propose a multistage *k*NN execution plan that enables high-performance *k*NN queries in distributed settings by overlapping the computation and communication stages. Moreover, we propose an adaptive data partitioning scheme that dynamically adjusts the workload among the operators according to the changes in the input values. Combining these two techniques, ADS-kNN provides a scalable and adaptive *k*NN join operator for data streams. Our experiments using a 56-core computer show that our ADS-kNN achieves a maximum throughput that is 21 times higher than that of a single-threaded approach.

Author's contributions: Conceived, developed, and implemented the approach. Devised optimizations. Conducted analysis and experimental evaluation. Wrote the paper.

4

Discussion

In this chapter, we discuss the approaches presented in this thesis in the context of other parallel computing methods for in-memory applications. We briefly describe the related work and highlight the differences and advantages of our solutions in comparison with the existing approaches.

To the best of our knowledge, our hybrid layout for B^+-Tree is among the first efforts to jointly leverage the performance capacities of both CPU and GPU memories. In comparison with previous heterogeneous implementations of B^+-Tree using an accelerated processing unit (APU) [42], our hybrid layout has two main advantages. First, our design is capable of utilizing two separate memory components simultaneously, system main memory and GPU memory, to achieve a higher combined memory bandwidth instead of relying on CPU memory. Second, our approach is capable of utilizing a discrete GPU, which offers relatively high computing power compared to an integrated GPU, as found in APUs. The performance of heterogeneous solutions utilizing an APU is bounded by the capabilities of the system main memory, which is a critical concern for data-intensive applications such as tree-based data indexing. Furthermore, the computing power of these integrated accelerators is not on par with that of a high-end discrete GPU or a FPGA.

Multiple approaches have been proposed to exploit the computing resources of a discrete

GPU for high-performance tree-based indexing [43, 44]. Awad et al. [16] proposed a dynamic B$^+$-Tree design optimized for a GPU-accelerated key-value store. FAST (Fast Architecture Sensitive Tree) is a hierarchical static binary tree that can be configured for various system configurations, such as a multicore CPU or GPU [45]. However, the indexing data structures proposed by these studies entirely reside in GPU memory and are therefore bounded by the limited capacity of GPU memory. In contrast, HB$^+$-Tree is capable of indexing datasets beyond the capacity of GPU memory by utilizing its hybrid layout.

Because of its computational complexity and importance in many data analytics scenarios, parallel window join has received considerable attention in recent years [46, 47, 48, 49, 50]. Handshake join is a distributed window join algorithm in which the tuple distribution is inspired by how soccer players perform handshakes [48]. It arranges processing cores in a chain layout and propagates the tuples of two given streams in the two opposing directions along this chain. Whenever two tuples reach each other at a point in the chain, the operating core performs an evaluation and decides whether the two tuples match each other. Though handshake join is a scalable approach, it suffers from high result latency. It may take a long time, depending on the window lengths, until two tuples meet each other in this chain. Roy et al.[49] improved handshake join by introducing a fast-forwarding mechanism that accelerates the transmission of each tuple toward its associated operating core. SplitJoin is another distributed stream join approach based on top-down tuple distribution [51]. It splits the join operation into two independent process and store subtasks to reduce the dependency between processing units. All these aforementioned approaches are based on context-insensitive partitioning in which tuples are distributed based on their arrival order rather than on their values. Although they are effective for nested loop join implementations, context-insensitive-based approaches do not perform well for index-based window join because of redundant index operations.

There is a wide body of work studying the problem of parallel data indexing using multicore CPUs [13, 52, 53, 54]. Though they are effective for a disk-resident data structure, concurrency control mechanisms based on coupled latching are known to suffer from high latching overhead and poor scalability for in-memory systems [55]. B-link tree is a modified B$^+$-Tree with a relaxed node structure designed for concurrent operations for in-memory systems [53]. Search operations in B-link tree are lock free,

and concurrency control is only needed for insert operations. Unlike approaches based on coupled latching, an insert operation in B-link tree holds only a single node lock at a time; it releases the parent node lock before acquiring its child node lock. Therefore, B-link tree does not suffer from coupled latching. However, there is a disadvantage in using the relaxed node structure of B-link tree, which may result in an inefficient state and a lower search performance in applications with high insertion rates. Bw-Tree is a latch-free variant of B^+-Tree, designed for multithreaded systems [14, 56]. To avoid locks, it uses a delta update mechanism and atomic pointer updates using compare and swap operations. Different from these approaches that handle concurrency at the tree node level, parallelism in PIM-Tree is based on the value distribution of concurrent operations and course-grained locking. As an advantage of our approach, update operations in PIM-Tree are as efficient as those in single-threaded B^+-Tree. To perform concurrent operations, the only overhead is to acquire a single mutex associated with each range of values.

To the best of our knowledge, our ADS-kNN is the first approach for distributed *k*NN join over data streams. Although there are several studies dedicated to distributed *k*NN join over static datasets, this problem is not addressed in streaming environments [24, 57, 58, 59]. Hadoop-GIS is a map-reduce-based spatial data warehousing system that extends Hive to support parallel spatial queries [57]. MD-HBase is an index layer for range and *k*NN queries based on HBase [24]. It transforms multidimensional data points into a one-dimensional space using a linearization technique and stores data points in a range-partitioned key-value store. Similarly, SparkGIS is an extension of Apache Spark for high-performance spatial queries such as range join and *k*NN join [60]. However, all these approaches are designed for prestored datasets, and they do not apply well to streaming environments. AQWA is an adaptive approach for spatial range and *k*NN join operators based on a map-reduce paradigm [59]. It distributes data into disjoint partitions using a *k*d-tree that adapts to changes in both data and query distributions. However, the adaptive data partitioning mechanism in AQWA is based on batch processing, which does not apply to streaming environments. Splitfire is a parallel algorithm for in-memory *k*NN self-join that replicates the potential *k*NN candidates into the neighboring partitions to improve the join performance [61]. Likewise, Splitfire is also based on data preprocessing, and therefore, it is not applicable to streaming applications, which is the central focus of our approach.

There is a group of works focused on kNN queries on streaming data [62, 63, 64, 65]. Koudas et al. [62] proposed an algorithm for approximate kNN query on a sliding window, which locates the k nearest neighbors within a given error bound. Mouratidis et al. [63] studied the problem of continuous nearest neighbor queries on data streams. They proposed a solution based on reducing nearest neighbor monitoring to the skyline maintenance problem and conceptual partitioning to reduce computational overheads. Yang et al. [64] proposed a high-dimensional R-tree (HDR-tree) to tackle the problem of the reverse kNN problem in streaming data. However, the scope of all these mentioned approaches is limited to single-threaded solutions, and they do not consider parallel computation.

5

Conclusions

In this thesis, we presented three solutions to three challenging problems in high-performance in-memory data processing to facilitate the adoption of modern hardware in data management systems.

First, we presented HB$^+$-Tree, a unique design for B$^+$-Tree, which is specifically tailored for a heterogeneous computing platform with a hybrid memory architecture. In contrast to previous approaches, which are intended to utilize an individual multicore CPU or a many-core GPU, HB$^+$-Tree is capable of jointly leveraging the hybrid memory architecture and the computing resources of heterogeneous CPU/GPU architectures. For this purpose, we proposed an indexing solution based on a hybrid memory layout and a heterogeneous search algorithm. Our hybrid memory layout distributes the tree nodes between CPU and GPU memories according to the frequency of access, and our heterogeneous algorithm jointly utilizes both the CPU and GPU to perform search queries. These two techniques enabled HB$^+$-Tree to achieve high-throughput search performance over large volumes of data.

To evaluate the advantage of our hybrid design for tree-based data indexing, we compared it against our CPU-optimized indexing data structure, which is a novel in-memory B$^+$-Tree utilizing various optimization techniques such cache blocking and SIMD-enabled parallelism. Our CPU-optimized B$^+$-Tree attains a 1.3 times higher throughput compared

with FAST, the fastest reported indexing performance for a comparable indexing data structure running on a single CPU. Exploiting the computing capabilities of a heterogeneous system, HB$^+$-Tree outperforms the CPU-optimized solution by a factor of 2.4 on average.

Second, we presented PIM-Tree, a novel data structure to tackle the challenges in parallel indexing of highly dynamic data. Unlike previous parallel indexing data structures that resolve concurrent operations at the tree node level, PIM-Tree divides indexed values into disjoint ranges and relies on query distribution to exploit parallelism. The combination of two techniques, range partitioning and delta update, enabled PIM-Tree to provide frequent update queries with a low concurrency overhead, which is highly required for data indexing in streaming applications. To complement our data structure, we introduced a parallel stream join algorithm on the basis of shared indexes to exploit the computing power of a multicore CPU.

The evaluation results using an octa-core processor indicated that our parallel window join algorithm utilizing PIM-Tree achieves an up to 5.6 times higher throughput compared with single-threaded window join. Furthermore, single-threaded window join using PIM-Tree resulted in a 60% higher throughput than that using B$^+$-Tree, which indicates the effectiveness of PIM-Tree in streaming applications.

Last, we introduced ADS-kNN, a distributed solution for *k*NN join over data streams based on adaptive space partitioning. The load distribution in ADS-kNN is based on an online workload analysis and repartitioning that does not require any prior knowledge about the input data distribution and better suits real-time applications. To this end, we proposed an adaptive data partitioning mechanism that constantly monitors and adjusts the load distribution among join-cores to maintain the system in an efficient state. Furthermore, we presented a multistage *k*NN query execution plan for scalable and low-latency *k*NN query execution.

Utilizing 52 join-cores, distributed stream *k*NN join using ADS-kNN resulted in a more than 30 times higher throughput than the single-threaded implementation, which indicates a parallelization efficiency of 57%. Furthermore, ADS-kNN achieved a 12 times higher throughput than distributed window join based on round-robin partitioning utilizing the same 52 join-cores.

Bibliography

[1] N. Khan, I. Yaqoob, I. A. T. Hashem, et al. "Big data: survey, technologies, opportunities, and challenges." In: *The scientific world journal* 2014 (2014).

[2] M. Chen, S. Mao, and Y. Liu. "Big data: A survey." In: *Mobile networks and applications* 19.2 (2014), pp. 171–209.

[3] Statista Inc. *Number of smartphones sold to end users worldwide from 2007 to 2020.* 2021. URL: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[4] Statista Inc. *Global digital population as of January 2021.* 2021. URL: https://www.statista.com/statistics/617136/digital-population-worldwide/.

[5] Statista Inc. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2024.* 2021. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

[6] John C. McCallum. *Memory Prices 1957+.* 2021. URL: https://jcmit.net/memoryprice.htm.

[7] Karl Rupp. *Microprocessor Trend Data.* 2020. URL: Trend%20Data.https://www.github.com/karlrupp/microprocessor-trend-data.

[8] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. "In-Memory Big Data Management and Processing: A Survey." In: *IEEE Transactions on Knowledge and Data Engineering* 27.7 (2015), pp. 1920–1948. DOI: 10.1109/TKDE.2015.2427795.

[9] G. Blake, R. G. Dreslinski, and T. Mudge. "A survey of multicore processors." In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 26–37.

[10] H. Kasim, V. March, R. Zhang, and S. See. "Survey on parallel programming model." In: *IFIP International Conference on Network and Parallel Computing*. Springer. 2008, pp. 266–275.

[11] A. J. Chris, L. Jan, and S. Noah. *CouchDB: The definitive guide.* 2010.

[12] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* 7th. Pearson, 2015. ISBN: 0133970779.

[13] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. "PALM: Parallel architecture-friendly latch-free modifications to B+trees on many-core processors." In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 795–806.

[14] J. J. Levandoski, D. B. Lomet, and S. Sengupta. "The Bw-Tree: A B-tree for new hardware platforms." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 302–313.

[15] S. A. McKee. "Reflections on the memory wall." In: *Proceedings of the 1st conference on Computing frontiers*. 2004, p. 162.

[16] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens. "Engineering a high-performance GPU B-tree." In: *Proceedings of the 24th symposium on principles and practice of parallel programming*. 2019, pp. 145–157.

[17] D. B. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[18] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. "Stream reasoning: A survey and outlook." In: *Data Science* 1.1-2 (2017), pp. 59–83.

[19] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter. "Real-time stream processing for Big Data." In: *it-Information Technology* 58.4 (2016), pp. 186–194.

[20] M. Stonebraker, U. Çetintemel, and S. Zdonik. "The 8 requirements of real-time stream processing." In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[21] P. M. Grulich, B. Sebastian, S. Zeuch, et al. "Grizzly: Efficient Stream Processing Through Adaptive Query Compilation." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 2487–2503.

[22] L. Golab, S. Garg, and M. T. Özsu. "On indexing sliding windows over online data streams." In: *International Conference on Extending Database Technology*. Springer. 2004, pp. 712–729.

[23] M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. "Towards scalable location-aware services: requirements and research issues." In: *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*. 2003, pp. 110–117.

[24] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. "MD-Hbase: A scalable multi-dimensional data infrastructure for location aware services." In: *2011 IEEE 12th International Conference on Mobile Data Management*. Vol. 1. IEEE. 2011, pp. 7–16.

[25] Z. Cheng and J. Shen. "Just-for-Me: an adaptive personalization system for location-aware social music recommendation." In: *Proceedings of international conference on multimedia retrieval*. 2014, pp. 185–192.

[26] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. "LARS: A location-aware recommender system." In: *2012 IEEE 28th international conference on data engineering*. IEEE. 2012, pp. 450–461.

[27] C. Xia, H. Lu, B. C. Ooi, and J. Hu. "Gorder: an efficient method for knn join processing." In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 756–767.

[28] S. Zeuch, B. D. Monte, J. Karimov, et al. "Analyzing efficient stream processing on modern hardware." In: *Proceedings of the VLDB Endowment* 12.5 (2019), pp. 516–530.

[29] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. "Rhino: Efficient management of very large distributed state for stream processing engines." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2471–2486.

[30] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1*. Tech. rep. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[31] R. Chandra, L. Dagum, D. Kohr, et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[32] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[33] L. Arge, G. S. Brodal, and R. Fagerberg. "Cache-Oblivious Data Structures." In: *Handbook of Data Structures and Applications* 27 (2004), pp. 7–1.

[34] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. "Software pipelining." In: *ACM Computing Surveys (CSUR)* 27.3 (1995), pp. 367–432.

[35] E. D. Demaine. "Cache-oblivious algorithms and data structures." In: *Lecture Notes from the EEF Summer School on Massive Data Sets* 8.4 (2002), pp. 1–249.

[36] J. Rao and K. A. Ross. "Making B+-trees cache conscious in main memory." In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 475–486.

[37] T. Jain and T. Agrawal. "The haswell microarchitecture-4th generation processor." In: *International Journal of Computer Science and Information Technologies* 4.3 (2013), pp. 477–480.

[38] R. Patel and A. Rajwat. "A survey of embedded software profiling methodologies." In: *arXiv preprint arXiv:1312.2949* (2013).

[39] R. A. Bridges, N. Imam, and T. M. Mintz. "Understanding GPU power: A survey of profiling, modeling, and simulation methods." In: *ACM Computing Surveys (CSUR)* 49.3 (2016), pp. 1–27.

[40] NVIDIA. *CUDA Toolkit Documentation Version 6.5: Profiler User's Guide*. 2015. URL: http://docs.nvidia.com/cuda/profiler-users-guide.

[41] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A portable programming interface for performance evaluation on modern processors." In: *The international journal of high performance computing applications* 14.3 (2000), pp. 189–204.

[42] M. Daga and M. Nutter. "Exploiting coarse-grained parallelism in B+tree searches on an APU." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 240–247.

[43] K. Kaczmarski. "B+-tree optimized for GPGPU." In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2012, pp. 843–854.

[44] J. Fix, A. Wilkes, and K. Skadron. "Accelerating braided b+tree searches on a GPU with CUDA." In: *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*. 2011.

[45] C. Kim, J. Chhugani, N. Satish, et al. "FAST: fast architecture sensitive tree search on modern CPUs and GPUs." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 339–350.

[46] B. Gedik, R. R. Bordawekar, and S. Y. Philip. "CellJoin: a parallel stream join operator for the cell processor." In: *The VLDB journal* 18.2 (2009), pp. 501–519.

[47] J. Karimov, T. Rabl, and V. Markl. "AJoin: Ad-Hoc Stream Joins at Scale." In: 13.4 (Dec. 2019), pp. 435–448.

[48] J. Teubner and R. Mueller. "How soccer players would do stream joins." In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 2011, pp. 625–636.

[49] P. Roy, J. Teubner, and R. Gemulla. "Low-latency handshake join." In: *Proceedings of the VLDB Endowment* 7.9 (2014), pp. 709–720.

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

[51] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. "SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision." In: *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016, pp. 493–505.

[52] R. Bayer and M. Schkolnick. "Concurrency of operations on B-trees." In: *Acta informatica* 9.1 (1977), pp. 1–21.

[53] P. L. Lehman and S. B. Yao. "Efficient locking for concurrent operations on B-trees." In: *ACM Transactions on Database Systems (TODS)* 6.4 (1981), pp. 650–670.

[54] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. "PLP: page latch-free shared-everything OLTP." In: *Proceedings of the VLDB Endowment* 4.10 (2011), pp. 610–621.

[55] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. "Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems." In: *VLDB*. Vol. 1. 2001, pp. 181–190.

[56] Z. Wang, A. Pavlo, H. Lim, et al. "Building a bw-tree takes more than just buzz words." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 473–488.

[57] A. Aji, F. Wang, H. Vo, et al. "Hadoop-GIS: A high performance spatial data warehousing system over MapReduce." In: *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*. Vol. 6. 11. NIH Public Access. 2013, pp. 1009–1020.

[58] A. Eldawy and M. F. Mokbel. "Spatialhadoop: A mapreduce framework for spatial data." In: *2015 IEEE 31st international conference on Data Engineering*. IEEE. 2015, pp. 1352–1363.

[59] A. M. Aly, A. R. Mahmood, M. S. Hassan, et al. "AQWA: adaptive query workload aware partitioning of big spatial data." In: *Proceedings of the VLDB Endowment* 8.13 (2015), pp. 2062–2073.

[60] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang. "SparkGIS: Resource aware efficient in-memory spatial query processing." In: *Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2017, pp. 1–10.

[61] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W.-C. Lee, and E. Pitoura. "Distributed in-memory processing of all k nearest neighbor queries." In: *IEEE transactions on knowledge and data engineering* 28.4 (2015), pp. 925–938.

[62] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. "Approximate NN queries on streams with guaranteed error/performance bounds." In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 804–815.

[63] K. Mouratidis and D. Papadias. "Continuous nearest neighbor queries over sliding windows." In: *IEEE transactions on knowledge and data engineering* 19.6 (2007), pp. 789–803.

[64] C. Yang, X. Yu, and Y. Liu. "Continuous KNN join processing for real-time recommendation." In: *2014 IEEE International Conference on Data Mining*. IEEE. 2014, pp. 640–649.

[65] K. Pripužić, I. P. Žarko, and K. Aberer. "Distributed processing of continuous sliding-window k-NN queries for data stream filtering." In: *World Wide Web* 14.5-6 (2011), pp. 465–494.

Appendix A

A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms

Amirhesam Shahvarani
Fakultät für Informatik
Technische Universität München
shahvara@in.tum.de

Hans-Arno Jacobsen
Fakultät für Informatik
Technische Universität München

ABSTRACT

An in-memory indexing tree is a critical component of many databases. Modern many-core processors, such as GPUs, are offering tremendous amounts of computing power making them an attractive choice for accelerating indexing. However, the memory available to the accelerating co-processor is rather limited and expensive in comparison to the memory available to the CPU. This drawback is a barrier to exploit the computing power of co-processors for arbitrarily large index trees.

In this paper, we propose a novel design for a B$^+$-tree based on the heterogeneous computing platform and the hybrid memory architecture found in GPUs. We propose a hybrid CPU-GPU B$^+$-tree, – HB$^+$-tree, – which targets high search throughput use cases. Unique to our design is the joint and simultaneous use of computing and memory resources of CPU-GPU systems. Our experiments show that our HB$^+$-tree can perform up to 240 million index queries per second, which is 2.4X higher than our CPU-optimized solution.

CCS Concepts

•Information systems → Data management systems; Data structures; •Computer systems organization → *Multicore architectures;*

Keywords

Heterogeneous Computing; Indexing; In-memory Database; B$^+$-tree

1. INTRODUCTION

The B$^+$-tree is a well known dynamic data structure, widely used as index in database management systems, data warehouses, online analytical processing (OLAP), decision support systems and data mining [10, 26, 4, 15]. Since the memory capacity of modern servers is sufficiently large, in many databases today, indexing information is kept in main memory in order to eliminate performance limitations arising from expensive disk I/O [35, 2]. Due to different characteristics of main memory, implementing an efficient in-memory B$^+$-tree involves different constraints [42].

Approaches that leverage GPUs to accelerate processing have become popular in many domains due to the superior computing power to price ratio offered by many GPUs [39, 41]. Also, in databases, several approaches have emerged to demonstrate the benefits of using GPUs to accelerate processing, such as, GPUTeraSort for sorting billion-record wide-key databases [17] and GPU-accelerated relational join processing [23][6]. Also, tree-based indexing, as a critical operation in databases, has been in the focus of recent approaches [27, 28, 13].

A GPU offers a higher memory bandwidth as compared to a CPU, which makes the GPU an attractive choice for database indexing. However, the efficient utilization of both memory bandwidth and computation resources of a GPU is a challenging endeavor because of distinct architecture of the GPU, which forces programmers to use the same arrangement of parallel threads for both computation and data transfer [7]. In addition, leveraging the GPU as a processing accelerator necessarily involves data transfer between main memory and GPU memory, resulting in additional latency [20].

In this paper, we present HB$^+$-tree (Hybrid B$^+$-tree), a modified B$^+$-tree, jointly leveraging CPU and GPU resources of the same compute platform. Our design is geared towards lookup intensive applications where tree updates are performed through bulk update processing, applicable to index updates in online analytical processing (OLAP), decision support systems and data mining [47, 48, 18].

Realizing indexing operations based on either CPU or GPU is subject to different trade-offs. CPU performance is bounded by *memory bandwidth* as the index grows beyond the size of the last level cache (LLC), while GPU performance is bounded by *memory capacity*. Although GPU's memory architecture is efficient enough enabling the GPU to reach higher throughput, the memory available to the GPU is more limited than CPU's main memory. Intuitively speaking, our design objective is to combine these characteristics of CPU and GPU memory to achieve high throughput for index tree operations over high volumes of data. We explore a hybrid design that scatters index data among CPU and GPU memory according to the volume and the frequency of accesses. Complementing this design, we proposed a heterogeneous CPU-GPU algorithm for searching the HB$^+$-tree. We develop a task pipelining method between CPU and GPU to

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882918

overcome the communication cost between them. For better computation resource utilization, the task pipeline is further extended by double buffering, which is a concurrency design pattern for avoiding delay in data transfer [19]. We also design a load balancing scheme to improve resource utilization for systems with different GPU to CPU computation power ratios.

Furthermore, we propose two versions of HB$^+$-tree in this paper. In addition to the *regular* HB$^+$-tree, capable of efficiently performing bulk updates, we propose an array representation, referred to as *implicit* HB$^+$-tree, which is more efficient for high-throughput search-only applications. Moreover, we develop a bulk update mechanism for the regular HB$^+$-tree, which deals with the challenges of utilizing a hybrid memory architecture. For both regular and implicit HB$^+$-tree, we develop and evaluate a 64-bit and a 32-bit key versions of the tree. The data structure and algorithm designs we describe are based on using 64-bit keys; the design differences for the 32-bit tree version are summarized at the end of each section.

To the best of our knowledge, HB$^+$-tree is among the first indexing approaches to *jointly* leverage the heterogeneous computing power of CPU and GPU as well as *jointly* utilize their separate memories to achieve a higher aggregate bandwidth than using either memory alone.

Our approach has two main advantages over previous heterogeneous implementations of B$^+$-tree employing, for example, an APU (Accelerated Processing Unit) [13]. First, our design benefits from the hybrid memory architecture to improve memory bandwidth instead of relying on the CPU's main memory alone. Heterogeneous platforms increase the potential computing power of a system, but applications which are bandwidth bounded cannot leverage the extra compute resources unless the memory bandwidth is also improved [29]. Second, we accelerate the index search using a discrete GPU which provides higher computing power than an integrated GPU, as found in APUs. An APU is a system processor equipped with additional processing resources such as a FPGA (Field-Programmable Gate Array) or an integrated GPU to accelerate a specific kind of computation. However, the computing power of these integrated components are not comparable to high-end discrete FPGAs or GPUs that are interconnected with the system via the PCIe bus.

We opted to develop our approach using CUDA, which is the widely-used parallel computing platform and programming model developed by Nvidia for general purpose computing on GPUs [44]. However, there is no technical limitation in our design that prevents porting our approach to other GPU platforms, such as OpenCL [46].

To highlight the advantages of our hybrid solution, we further develop a CPU-optimized, multi-threaded B$^+$-tree, as baseline for comparison with our HB$^+$-tree. For this design, we also develop regular and implicit as well as 64-bit and 32-bit tree versions.

To ensure our CPU-optimized B$^+$-trees exhibits adequate performance, we also implement FAST – the fastest reported indexing performance of a comparable solution running on a single CPU [29] – and compare our implementations against it. Our CPU-optimized B$^+$-tree attains 1.3X higher throughput than FAST on average. Furthermore, for our CPU-optimized tree, we propose a novel tree structure optimization based on cache blocking and SIMD-enabled parallel search algorithm, and show how the use of huge pages help to increase the throughput of index search operations using a single CPU. Several components of the CPU-optimized B$^+$-tree are used in the implementation of our HB$^+$-trees.

We evaluate our solutions for varying number of tuples from 8M to 1B and show how each design decision affects the indexing performance.

HB$^+$-tree achieves up to 240 and 210 million queries per second for implicit and regular tree versions, respectively, which is 2.4X times higher than the results for our CPU-optimized B$^+$-tree.

The remainder of the paper is organized as follows. Section 2 surveys related approaches. Section 3 provides background information on B$^+$-tree. Section 4 presents our CPU-optimized design and implementation of B$^+$-tree. Section 5 introduces our HB$^+$-tree, including implementation details. Section 6 presents our experimental evaluation. Section 7 gives our conclusions and identifies future works.

2. RELATED WORK

A large body of work has been developed to optimize B-tree-like indexing. In this section, we focus on analyzing related work on in-memory indexing employing the power of parallel computing platforms.

B$^+$-tree is an indexing structure originally designed for systems with small main memory and comparatively large hard disks [8][15]. To optimize for costly disk I/O, B$^+$-tree operations are performed for entire disk blocks yielding fewer I/O transactions.

Flash-aware indexing trees such as BF-tree, FD-Tree, and LA-Tree have been proposed to reap performance benefits from the superior bandwidth and latency of solid state drives [5, 34, 1]. Furthermore, there exist many approaches for in-memory indexing in order to exploit the superior bandwidth and latency of system main memory. For example, Zhang et al. [22] provide a comprehensive review of data structures for in-memory data management such as for time-/space efficient indexing and concurrency control.

T-tree was proposed for databases where both indexing information and data records reside in main memory [31]. Lu et al. [36] showed that B$^+$-tree outperforms T-tree when concurrency control mechanisms are enforced. Rao et al. [42] introduce a cache-conscious indexing data structure, called Cache Sensitive Search Tree (CSS-tree), which is designed for predominantly static data. Later, Rao et al. [43] extended CSS-tree to CSB$^+$-tree to support incremental updates.

The Bw-tree is designed to exploit the caches of modern multi-core chips and the superior bandwidth of flash storage [33]. Zhou et al. [50] present an access buffering technique for in-memory tree-structured indexes that avoids cache thrashing. Mao et al. [37] introduced Masstree, a shared concurrent data structure combining B$^+$-tree and tries tailored to multi-cores. Hankins et al. [21] studied the effect of node size on cache misses, instruction count, and TLB misses for the CSB$^+$-tree. Based on their experiments, using nodes with sizes of 512 bytes and above, resulted in fewer TLB misses and better performance, while setting nodes size equal to the cache line width produced fewer cache misses but higher TLB misses. Chen et al. [11] explored how prefetching could improve operations in B$^+$-tree, also concluding that nodes wider than a single cache-line resulted in better performance. ART (Adaptive Radix

Tree) and FAST (Fast Architecture Sensitive Tree) are the latest data structures targeting high throughput in-memory indexing [32] [29]. ART is an adaptive radix tree (trie) for high speed in-memory indexing which exhibits better memory usage than previous radix trees. Alvarez et al. [3] compared the lookup throughput and memory footprint of ART to different data structures including B$^+$-tree and hash indexes. FAST is a static binary-tree developed for multi-core systems, which is configurable according to system characteristics such as cache-line size, memory page size and SIMD width. Sewall et al. [45] introduced PALM, a parallel latch-free modification of a B$^+$-tree designed for multi-core processors which is capable of concurrent search and update processing.

All these approaches are developed to utilize either a single-core or a multi-core CPU expect FAST which is capable to be configured for many-core GPU accelerators. But it is only able to operate on GPU resident data and assumes that the data fits into the GPU memory; it is therefore bounded by the GPU memory capacity. There are other GPU-accelerated index structures, which suffer from the same limitations. Fix et al. [16] presented an approach for a GPU-accelerated B$^+$-tree by proposing to modify the memory layout of the B$^+$-tree optimized for GPU memory. No GPU search throughput is reported, but a 9.4X to 19.2X speedup over a single-threaded CPU implementation is shown. Kaczmarski [27] proposed a GPU-specific implementation of B$^+$-tree which is capable of performing efficient updates. Although this approach performs bulk insertions faster than a CPU implementation, search throughput does not surpass 25.6 Kilo Queries Per Second (including copying keys from CPU to GPU and returning values back). Also in [28], the authors proposed a p-ary search with the goal of improving response time of query search using GPUs.

The limited capacity of GPU memory is addressed by other approaches. Daga et al. [13] utilize an APU (Accelerated Processing Unit) to accelerate search in a B$^+$-tree. Since the integrated many-core processor is directly accessing system main memory, their approach does not suffer from the penalty of having to move data between CPU and GPU memory and the limited capacity of GPU memory does not constitute a problem. However, it is still bounded by system main memory bandwidth, which is a critical problem for tree traversal as the tree grows [29]. Their implementation achieved up to 18 MQPS for a 6-core CPU and 70 MQPS for an APU (operating on system memory). Although, an APU is a heterogeneous computing platform, our solution based on a similar platforms has two main advantages. First, we are jointly utilizing two memory components, which is critical to improve the overall system memory bandwidth, while the APU approach is still relaying only on system main memory. Second, the computing power of high-end discrete GPUs is significantly higher than the integrated accelerator available in APUs.

3. B$^+$-TREE BACKGROUND

B$^+$-tree is a variation of B-tree which stores values only in leaf nodes, while inner nodes only comprise keys [15]. Hence, inner nodes and leaf nodes are represented by different data structures. Beside the characteristics adapted from B-tree such as height-balance and optimized memory access, B$^+$-tree offers faster range query support because of its sorted linked leaf nodes. The branching factor of the inner nodes

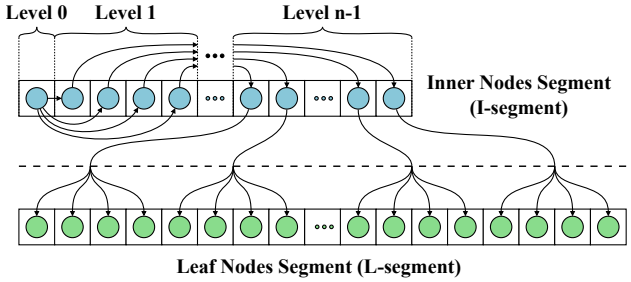


Figure 1: Arrangement of nodes in I-segment and L-segment.

is called the *order* of the B$^+$-tree. Inner nodes of order m, store up to $m-1$ keys and m child references.

Search in B$^+$-tree is a step-wise process, traversing the tree from the root node, each step consists of two parts: first, the search detects the child node associated with an interval which holds the targeted key, and, second, traversal proceeds to the next node. The process continues until the target key-value pair is found in a leaf node. To perform a range query on a B$^+$-tree, one can simply search for the first key in the range and then traverse leaf nodes until the last key is found.

Data structures in which the structural information is implicitly preserved in the way data is stored rather than explicitly through pointers, are called *implicit data structures* [38]. In implicit representation of B$^+$-tree, nodes are arranged in a breadth-first fashion in a one dimensional array. Since a node's child locations are known, and there is no need to store pointers, an implicit B$^+$-tree requires less memory and provides higher search throughput as compared to a regular B$^+$-tree. However, using an implicit representation leads to a linear time penalty for insert and delete operations. To distinguish the implicit representation from the one with pointers, we refer to the latter as the *regular* B$^+$-tree and the former the *implicit* B$^+$-tree in the rest of this paper.

The notations we use in this paper is summarized below.

H	: Height of root node (leaves are at height zero).
S	: Size of a variable (a key or a value) in bytes.
S_I	: Size of an inner node in bytes.
S_L	: Size of a leaf node in bytes.
F_I	: Maximum fanout of an inner node.
P_L	: Maximum capacity of key-value pairs in a leaf node.

4. CPU-OPTIMIZED B$^+$-TREE

In this section, we describe our parallel design of both, the implicit and the regular B$^+$-tree, optimized to exploit the features of a multi-core Intel CPU. Our CPU-optimized solutions serve as baseline in the evaluation of our hybrid solution, the HB$^+$-tree, described in the next section. The three main optimization we applied for the CPU-optimized solutions are : (1) an SIMD-enabled search algorithm based on the Intel AVX extension, (2) cache blocking to minimize cache misses, (3) huge page utilization to reduce TLB misses.

4.1 Tree Layout

The node structures of the CPU-optimized B$^+$-tree are designed with regards to minimizing both cache and TLB misses during search operations. We make use of huge pages by developing our own memory allocator which allows deter-

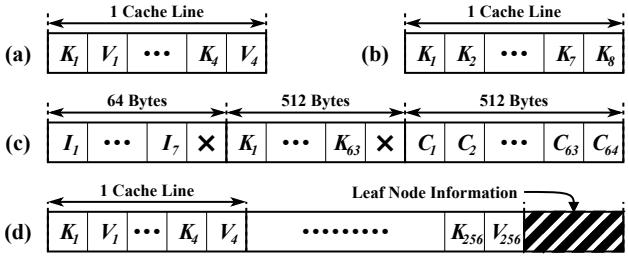


Figure 2: Node structure for CPU-optimized B$^+$-tree. (a) leaf node on implicit B$^+$-tree (b) inner node of implicit B$^+$-tree (c) inner node of regular B$^+$-tree (d) leaf node of regular B$^+$-tree (× indicates the element in fixed to maximum value).

mining whether a node resides on a huge page or not. Coupled with our tree node segmentation, which separates inner and leaf nodes into different segments, our approach minimizes the cost of TLB misses. We also use cache-conscious node structures for better cache data utilization.

The nodes are split into two segments: Inner node segment (I-segment) and leaf node segment (L-segment). The I-segment is always allocated to huge pages, while the L-segment could be allocated to either a huge page or a 4KB page, depending on the total size of the B$^+$-tree. For a given set of N tuples, the space needed for the I-segment (I_{space}) and L-segment (L_{space}) is given in Equation 1 (assuming the tree is full). Since there are only four entries in the last level TLB for 1GB pages and to assure that accessing inner nodes cause no TLB misses, the I-segment must not be larger than 4GB.

$$I_{space} = \frac{N}{P_L(F_I - 1)} \times S_I, \; L_{space} = \frac{N}{P_L} \times S_L \qquad (1)$$

Query search starts from the I-segment, where the root resides, and after passing all inner nodes, continues in the L-segment to determine the target key-value pair in the leaf.

The total number of TLB misses depends on whether the L-segment is placed in a 1GB or a 4KB page. In case of using a 4KB page, since accessing the I-segment causes no TLB miss and each leaf node resides within its individual 4KB page, there is at most a single TLB miss per lookup. If the required memory to store both segments is not more than 4GB, the best option is to also allocate the L-segment on the huge page. Using such a configuration causes no TLB miss for the entire search operation. If the size of the tree exceeds 4GB and the L-segment is allocated to a huge page, the total number of misses depends on the sequence of input queries and the TLB replacement policy.

We design different inner node data structures for our implicit and regular B$^+$-tree, as detailed in Figure 2.

Implicit B$^+$-tree: Since in this tree organization, nodes are arranged in a breadth first fashion, the child node locations are implicitly known. If the node A is the i_{th} node of a tree at level m in breadth first order, then the j_{th} child of A is at position $Offset[m+1]+i\times F_I+j$, where $Offset[l]$ is pointing to the beginning of the l_{th}-level. As a result, it is possible to achieve a higher fanout using the same amount of memory in comparison to the regular B$^+$-tree. We dedicate one cache line per each inner or leaf node ($S_I = S_L = 64$). The only content of leaf nodes are key-value pairs as it shown in Figure 2(a). Since all nodes are fully occupied and they are

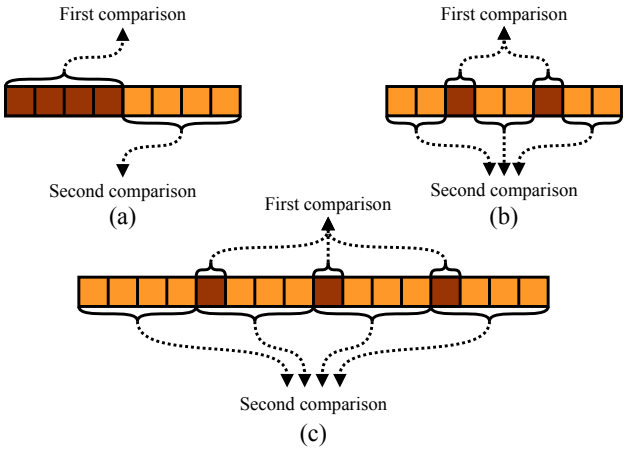


Figure 3: Node search using AVX unit. (a) linear 64-bit (b) hierarchical 64-bit (c) hierarchical 32-bit.

placed in order, there is no need to explicitly maintain node size as well as next and previous node pointers for the linked list of leaves. Each inner node filled with eight keys as illustrated in Figure 2(b). The number of cache lines required per search query is $H + 1$, where $H = \lceil \log_9 (N/4 + 1) \rceil$.

Regular B$^+$-tree: The minimum amount of space required for an inner node with m children is equal to $(m - 1)S + mP$ bytes, where P is the reference size in bytes. Considering $S = P = 8$, the maximally achievable fanout using a single cache line is limited to 4. Such a small fanout leads to many random memory accesses during search operations. For better lookup performance, we propose a structure for inner nodes consisting of indexes, keys and child references. As illustrated in Figure 2(c), each inner node consist of 17 cache lines ($S_I = 1088$), where the first one is dedicated to indexes, while keys and references are arranged in the following sixteen cache lines ($F_I = 64$). Each index is assigned to the maximum value of the corresponding cache line ($I_s = K_{8s}$). Utilizing indexes, only three cache lines are retrieved to find the successor node. The search algorithm first searches indexes and, based on the comparison result, fetches the corresponding cache lines, which includes the targeted child reference.

We apply inner node fragmentation in order to achieve better memory and cache line data utilization. The data of each inner node is broken up into two fragments. One fragment contains key-value pairs and child references, and the other one contains the node size, parent and sibling references. Whenever an inner node is needed, our node memory allocator dedicates one of each fragment from two separated data structures in such a way that both fragments share the same index which can be used to retrieve both fragments later on. Also, we set all empty keys of each inner node to the maximum value in our implementation ($2^n - 1$ for an n-bit integer), so that the lookup algorithm is able to perform node search without knowing the inner node size.

The size of a leaf node in the regular B$^+$-tree impacts the range query performance. Moving to the successor leaf node in the implicit B$^+$-tree can be done very efficiently, as leaf nodes are arranged sequentially. But the small leaf node capacity of the regular B$^+$-tree causes a series of cache misses during range query execution and, thus, decreases

performance, while leaf nodes bigger than a cache line lead to slower leaf node search. To address this problem, we designed bigger leaf nodes and make use of a dedicated memory pool manager for allocating leaf nodes and last level inner nodes so that both point and range queries can be realized efficiently. We pack 64 small leaf nodes into a bigger node, which we extend with another cache line to store leaf node information. Each last level inner node is only related to one big leaf node. Similar to the node fragmentation technique we used, here, our memory pool manager allocates leaf nodes and last level inner nodes from two different memory pools in such a way that both nodes share the same index. Consequently, the tree lookup algorithm can directly retrieve the cache line in the leaf node, where the targeted key is located, by using the index of the last inner node and the inner node search result. Moreover, we set all empty elements of a leaf node to the maximum value, which enables the lookup algorithm to search a leaf node without knowing the size of the leaf node. In case the search key is the maximum value, the lookup algorithm must read the node size to perform leaf node search. Although the capacity of the bigger leaf node is 256 key-value pairs, we consider $P_L = 4$ in our analysis since the addressable units from a last inner node are cache lines with a capacity of 4. The structures of implicit and regular B$^+$-tree are illustrated in Figure 2(a) and (d), respectively. The total number of cache lines needed for each query is $3H + 1$, where H (height of tree) is:

$$\left\lceil \log_{32}\left(\frac{N}{4}+1\right)\right\rceil \leq H \leq \left\lfloor \log_{16}\left(\frac{N/2+1}{2}\right)\right\rfloor + 1 \qquad (2)$$

Using 32-bit variables, 16 keys or values can fit into a cache line. Consequently, an inner node's fanout increases to 17 and 256 for implicit and regular B$^+$-tree, respectively. The capacity of each cache line in leaf nodes increases to 8.

4.2 Utilizing SIMD Unit for Search

The Advanced Vector Extensions 2 (AVX2) is the latest enhancement to Intel x86 processors for SIMD operations. AVX2 is capable of operating on 256-bit registers, which is equivalent to eight 32-bit or four 64-bit integers.

Since the size of AVX registers is half the size of a cache line, it is not feasible to compare an entire cache line using a single AVX comparison operation. We propose two different approaches to employ the AVX unit: linear and hierarchical.

The linear approach divides the cache line into two equal parts and separately searches each one. In contrast, the hierarchical approach divides the array into three equal parts and uses the boundary keys to locate the part where the target is placed.

The hierarchical approach needs less data loaded into AVX2 registers, while the linear approach is control dependency free, which is safe for out-of-order execution. We also implemented sequential search as a baseline to measure the resulting speedup. Our AVX-enabled search algorithms are illustrated in Figure 3.

Software Pipelining is a method to improve loop performance by rearranging instructions such that the instructions of the modified loop are chosen from different iterations of the original loop [24]. Using this method, dependent instructions from a single iteration are scattered among multiple loop iterations, so that the CPU pipeline can be scheduled to reduce instruction stalls caused by memory latency. To this end, each CPU thread loads a batch of queries and resolves them concurrently. Using this configuration, the thread switches to resolving another query whenever the current search operation is blocked by a data access. The optimal size for batches depends on the system configuration. Small batch sizes cannot provide reasonable overlap, while large overlap leads to inefficient CPU register utilization. In our experiments, a size of 16 resulted in the best performance. The total number of concurrent queries is up to $16 \times CPU\ Threads$.

5. HYBRID CPU-GPU B$^+$-TREE

In this section, we introduce the design of our CPU-GPU hybrid B$^+$-tree. We describe the tree's memory layout and the heterogeneous CPU-GPU search algorithm. Finally, we present a load balancing method, as technique for fine-tuning our hybrid tree across systems with different GPU-to-CPU computation power ratios.

5.1 Overview

Current multi-purpose processors are heavily relying on cache units to mitigate the memory wall problem [49]. For trees which would fit entirely into the last level cache (LLC), caching is very effective and memory latency would almost vanish. However, search throughput drops noticeably as the tree size surpasses LLC capacity and becomes memory bound [29]. Although techniques such as prefetching and software pipelining are applicable for tree-based index search to alleviate the memory latency problem, the system performance is still bounded by the memory bandwidth [11, 29].

The results from previous efforts of implementing a B+-tree on GPUs demonstrate the realizable performance benefits [29]. Instead of relying on caching, GPUs use high degrees of multi-threading and fast context switching logic with near zero overhead, to hide memory latency [12]. Since this mechanism is not affected by the volume of data, the throughput of tree indexing using GPUs is more resilient against tree growth. As result, GPU-based approaches outperform CPU-based approaches for tree sizes larger than the LLC [29]. However, GPUs cannot maintain their performance advantage, because the amount of their memory is limited in comparison to CPUs. It is not feasible to make use of the computational capabilities of GPUs, when the tree grows beyond the GPU memory capacity using previous methods [13].

To address this dilemma, we propose a new B$^+$-tree, called HB$^+$-tree, leveraging the hybrid memory architecture and heterogeneous computing model of today's computing platforms. Here, we employ the computing power of discrete many-core accelerators for index searching on trees larger than the accelerator's dedicated memory. We design HB$^+$-tree based on a compute platform accelerated by GPUs. To achieve higher total memory bandwidth, we scatter the nodes across GPU *and* CPU memory in a way which enables the index search algorithm to utilize *both memories* concurrently. As a result, the effective system memory bandwidth is the aggregate of both memory units. Also, we design a heterogeneous search algorithm to minimize the communication overhead between processors and *utilize both* – GPU and CPU – simultaneously.

Since our target use cases are lookup-intensive and batch update processing dominated scenarios (e.g., data ware/-

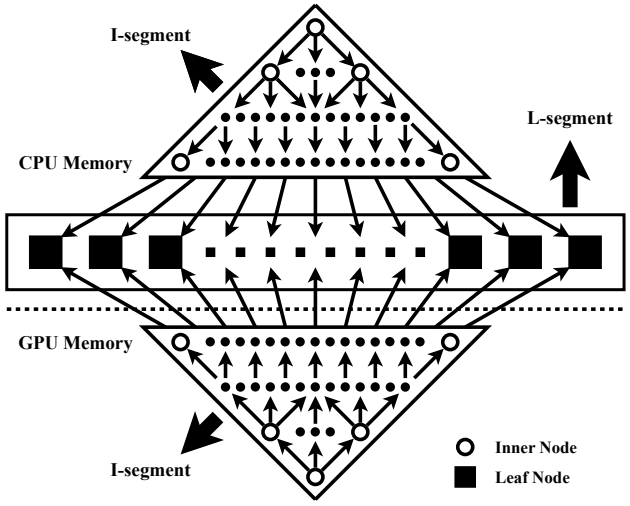


Figure 4: HB$^+$-tree node arrangement: The triangular area is the I-segment which is duplicated on both CPU and GPU memory and the rectangular area is the L-segment which resides only in the CPU memory.

houses), we envision that our indexes are integrated into existing systems based on passing query input and index output via CPU main memory. Also, all our HB$^+$-tree versions exhibits the same interface as our CPU-optimized B$^+$-tree; both follow the conventional B+-tree interface.

5.2 Tree Layout

Similar to the CPU-optimized tree, HB$^+$-tree also consist of I-segments and L-segments. The L-segment is configured based on the CPU search algorithm and only resides in CPU memory, while the I-segment resides in both GPU and CPU memory, i.e., it is mirrored across both memory units. The rationale for this design is that leaf nodes require more space than inner nodes for storage and are less frequently accessed; thus, we place them in CPU memory, which has a higher capacity but lower bandwidth.

Figure 4 illustrates the placement of inner and leaf nodes in the HB$^+$-tree. The leaf nodes of both, the regular and implicit HB$^+$-tree, are identical to the ones of the CPU-optimized version of the tree.

Unlike main memory, the GPU memory architecture does not have a fixed unit of transfer. As a warp executes an instruction accessing GPU memory, the GPU translates the access into one or more aligned data transfers of size 32, 64 or 128 bytes [40]. This limitation is a consequence of coalesced memory access, which results in higher bandwidth as well as higher latency.

We discovered that the best balance between thread scheduling efficiency and bandwidth utilization results from using transfers of size 64 bytes. Since our CPU-optimized B$^+$-tree nodes are also based on 64 byte transfers, the inner node structures of HB$^+$-tree are similar to our CPU-optimized B$^+$-tree. For the regular version, the inner nodes are identical, but we reduce fan-out of inner nodes in implicit HB$^+$-tree to 8, so that we can utilize the same thread hierarchy for both data access and node search and avoid warp divergence, and we set the last key (K_8) to the maximum representable value.

For the 32-bit version, F_I is increased to 16 and 256 for implicit and regular HB$^+$-tree, respectively.

5.3 Parallel Node Search on GPU

In this section, we first describe our search algorithm for an arbitrary sized array and then explain how it is used for search in HB$^+$-tree.

For a given key and a sorted $array$ of s elements such that key is not bigger than the last element ($key \leq array[s]$), the parallel search algorithm finds the maximum index i such that $key \leq array[i]$. The possible values for i are $[1..s]$. To find the target index i, the search algorithm initializes s threads ($t_j : 1 \leq j \leq s$), where each thread is assigned to a single result value. First, each thread (t_j) compares key to the associated value ($array[j]$) to check whether key is less than or equal to $array[j]$ and stores the result ($r_t : 0, 1$) in a shared array. Based on the thread's local comparison result (r_t) and the result from the prior thread (r_{t-1}), each thread determines if it is assigned the final answer. If so (i.e., $r_t = 1$ and $r_{t-1} = 0$), the thread sets the final answer to its own index.

Because the last keys of all inner nodes of HB$^+$-tree are always set to the maximum ($2^n - 1$ for an n bit number), it is assured that all queries are less than or equal to the last key, and our search algorithm always returns a valid result.

Searching an inner node in the regular HB$^+$-tree is slightly different and requires three memory accesses instead of one and involves three steps. First, the parallel search algorithm is applied on indexes to determine the interval of keys containing the search query. Then, the corresponding interval is fetched from GPU memory and searched using the parallel algorithm to identify the next node position. Finally, the address of the next level node is retrieved using an extra memory transfer.

The total number of concurrent queries at the GPU is equal to $GPU\_Threads/T$, where the optimal number of $GPU\_Threads$ depends on the GPU specification and T is the number of threads dedicated per each query (8 for a 64-bit implementation and 16 for a 32-bit implementation).

5.4 Search Query Execution

Since we considered that the input queries are given in CPU memory, the first step is to transfer them into GPU memory, before the GPU starts executing a search operation. After GPU finished its task, the intermediate results, – references to nodes where the search operation must be resumed, – are transfered into main memory after the GPU completes the search operation. In the last step, the CPU continues the search operation to reach the target tuple. The execution of a search on the CPU is analogous to the implementation for the CPU-optimized B$^+$-tree.

The given queries are broken into buckets of size M which are processed independently according to the following steps, where $T_i, i = 1..4$ are times required for each step in our cost model.

1. Transfer bucket to GPU memory.
   $T_1 = T_{init} + (M \times S)/Bandwidt$
2. GPU traversal of all inner nodes of tree per each query.
   $T_2 = K_{init} + (M/SIMD_G) \times P_{GPU}$
3. Transfer of intermediate results to CPU memory.
   $T_3 = T_{init} + (M \times R)/Bandwidt$
4. CPU continues search in leaf nodes.
   $T_4 = (M/SIMD_C) \times P_{CPU}$

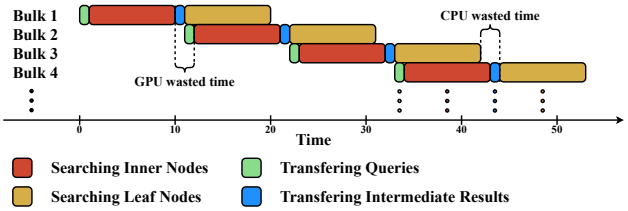


Figure 5: CPU-GPU pipelining.

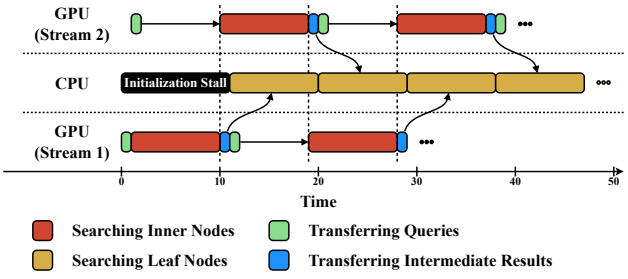


Figure 6: CPU-GPU pipelining with double buffering.

R	: Size of an intermediate result in bytes.
T_{init}	: Data transfer initialization time between main memory and GPU memory.
K_{init}	: GPU initialization time for search operation.
$SIMD_G$	: GPU SIMD width.
$SIMD_C$	: CPU SIMD width.
P_{GPU}	: Average processing time for a query on GPU.
P_{CPU}	: Average processing time for a query on CPU.
$Bandwidth$	: Data transfer bandwidth between main memory and GPU memory.

Assigning the proper value for M is important since both performance parameters, throughput and latency, are controlled by M. Small bucket sizes increase the influence of overhead constants (K_{init} and T_{init}) against effective computation time, leading to lower throughput; increasing M increases the cost of each step (T_i), resulting in higher latency.

Apart from how each bucket is processed, bucket scheduling is also important for optimal utilization of resources. The simplest approach is to load and resolve each query bucket sequentially. The drawbacks of using this approach are two-fold: (1) it is not feasible to utilize both processors concurrently, and (2) there is no opportunity of overlapping communication and computation to eliminate data transfer overhead. Thus, the cost for resolving each bucket is the aggregate of all steps ($T_S = \sum_{i=1}^{4} T_i$). We propose CPU-GPU pipelining and employ a double buffering technique to eliminate these drawbacks.

CPU-GPU pipelining improves system performance by overlapping the execution of buckets. As illustrated in Figure 5, the next bucket is loaded as soon as the intermediate result of the current bucket is transferred into CPU memory. In this way, CPU and GPU can be utilized concurrently. The average time needed to resolve queries within a bucket is reduced to $T_P = T_1 + \max(T_2 + T_3, T_4)$ (ignoring pipeline initialization stalls). Considering $T_2 = T_4$, T_P is equal to $T_1 + T_2 + T_3$, then CPU processing time has been eliminated.

Furthermore, we extend pipelining with double buffering to eliminate the data transfer time. The timeline for the enhanced pipelining approach is given in Figure 6. We initiate two GPU threads which are working on separated buffers but share the same processors, where each thread operates as a CPU-GPU pipelined approach. The average cost of processing each bucket is $T_P = \max(T_2, T_4)$, considering that data transfer time is smaller than computation time.

Although double buffering improves overall system throughput, it also increases processing latency because of prefetching of buckets. The average latency of the pipelined approach is $T_1 + T_2 + T_3 + \frac{T_4}{2}$, which increases to $2 \times T_2 + \frac{T_4}{2}$ by applying double buffering.

5.5 Load Balancing Scheme

Our HB$^+$-tree design is primarily targeting systems which are accelerated using sufficiently powerful GPUs and the system throughput is bounded by the CPU. Therefore, HB$^+$-tree devotes only a small share of the query load to the CPU, which is only searching leaf nodes while all inner nodes are processed by the GPU.

To offer a more generally applicable solution, we enhance HB$^+$-tree with a load balancing mechanism, which improves resource utilization on systems with an arbitrary GPU-to-CPU computation power ratio, refereed to as *load balanced* HB$^+$-tree.

With the load balancing scheme, the CPU starts traversing inner nodes up to a specific depth (D) and transfers the query and the intermediate inner node index to GPU memory. Then, the GPU resumes traversing up to the final inner node level and returns the leaf node index to the CPU. Finally, the CPU searches the leaf node to determine the target key-value pair. We prefer to dedicate the top inner nodes to the CPU since the space required for them is comparably lower than the inner nodes at the bottom of tree resulting in better cache utilization and lookup performance.

Let $I_{G,i}$ and $I_{C,i}$ be the average cost of searching at depth i for GPU and CPU, respectively, and let L_C be the average cost of searching a leaf node, then the average cost of a single search (C) is given according to Equation 3. Adjusting the parameter D is required to minimize C_{inner}.

$$C = \max(L_C + \sum_{0}^{D} C_{C,i}, \sum_{D+1}^{H} C_{G,i}) \tag{3}$$

Moreover, to provide a finer granularity for work load distribution, we divide each bucket into two parts. For the first part, $R \times M$ queries ($0 \leq R \leq 1$) of a bucket, the CPU searches only D levels of inner nodes, while for the rest of the queries ($M \times (1 - R)$), the CPU searches $D + 1$ levels. Using the new parameter R, the search cost C is updated to Equation 4.

$$C = \max(L_C + \sum_{0}^{D-1} C_{C,i} + R\, C_{C,D}, (1-R) C_{G,D} + \sum_{D+1}^{H} C_{G,i}) \tag{4}$$

We develop a discovery algorithm to determine the values for D and R that minimize C. The algorithm starts from $D = 0$ and $R = 1$, where it dedicates the maximum possible load to the GPU. First, it linearly searches for the optimal value of D (coarser parameter). Then, it adjusts R (finer

parameter) using binary search. The discovery algorithm is given in Algorithm 1.

Algorithm 1 Discovery algorithm

```
1: D ← 0, R = 1
2: (Time_GPU, Time_CPU) = getSample(D, R)†
3: while Time_GPU > Time_CPU do
4:     D ← D + 1
5:     (Time_GPU, Time_CPU) = getSample(D, R)
6: R ← 0.5
7: for step ← 2 to 5 do
8:     (Time_GPU, Time_CPU) = getSample(D, R)
9:     if Time_GPU > Time_CPU then
10:        R ← R + 1/(2^step)
11:    else
12:        R ← R − 1/(2^step)
```

†*getSample* runs the program for given D and R; it returns the time GPU and CPU require to perform their work share.

We also change the bucket handling strategy which is advantageous only for GPU bounded systems. The GPU must perform thread scheduling prior to starting effective kernel execution as a new kernel program is submitted to the GPU. Pre-submitting of a successor kernel before the current one is finished, enables the GPU to perform scheduling of the next kernel, concurrently to the previous kernel execution. For this to work, we require at least three concurrently operating buckets. Since this optimization technique is not effective for CPU bounded systems, we restrict the number of query buckets in the not-load-balanced version of HB$^+$-tree to two in order to reduce latency. However, we increase the number of query buckets to three in the load balanced implementation of the HB$^+$-tree for better GPU utilization.

5.6 Batch Update

The implicit B$^+$-tree is not capable of processing individual updates. Whenever an update is required, the entire tree must be re-built. The algorithm first builds both I-segment and L-segment in main memory based on the new dataset and, subsequently transfers the I-segment to GPU memory.

Efficiently processing concurrent batch updates with the regular HB$^+$-tree faces two challenges: (1) I-segment synchronization and (2) concurrency handling. The former is specific to HB$^+$-tree, while the latter is a general challenge for tree indexing. We propose two different tree update methods; their performances depends on the batch size.

We design an asynchronous parallel update method which first performs updates in main memory in parallel and then transfers the entire I-segment to GPU memory. The given update queries are processed in groups of size 16K. Each thread takes a query and searches the tree up to the last level inner node. At this point, the thread checks if the query execution causes any node merge or split. If not, it requests the lock assigned to the inner node and performs the update. Because of HB$^+$-tree's big leaf nodes (256 entries), more than 99% of the update queries can be resolved this way, on average. The remaining unresolved queries are processed subsequently using a single thread. When all queries are executed, the I-segment in GPU memory is updated. This method is more efficient for bigger batch sizes which often result in many inner node modifications. In these cases, it is more beneficial to transfer the entire I-segment once, instead of performing many small transfers for each inner node.

For smaller batch sizes, we propose a synchronized update method which is performed by two threads, a *modifying* and a *synchronizing* one. The modifying thread executes update queries and submits a request for each modified inner node to a shared queue. Upon receiving a request, the synchronizing thread updates the inner node in GPU memory according to the node's replica in main memory. Using this method, tree update and node synchronization proceed concurrently. Although, it is feasible to implement this method with multiple modifying and synchronizing threads, we found the performance of this method is bounded by the communication initialization latency between main memory and GPU memory which was not reduced by parallelism.

6. EVALUATION

We now present the performance evaluation of both CPU-optimized B$^+$-tree and HB$^+$-tree. First, we describe the experimental setup and workload. Then, we demonstrate the impact of various optimizations on the individual approaches, and finally, we compare the search operation performance of CPU-optimized B$^+$-tree and HB$^+$-tree considering latency and throughput.

6.1 Experimental Setup

We used two system setups for evaluating our approaches. The first machine (M_1) is equipped with Intel Xeon E5-2665 accelerated by the Nvidia Geforce 780 GTX. The second machine (M_2) is an Intel Core-i7 4800MQ accelerated by the Nvidia Geforce 770M GTX.

For all experiments except the experiment on skewed data, we generated multiple sets of key-value with 8M (2^{23}) to 1B (2^{30}) tuples, where keys and values are randomly generated according to a uniform distribution on $[0 - MAX]$ ($MAX = 2^n - 1$, n is number of bits: 32 or 64). After constructing the B$^+$-tree using this set, we randomly permuted the pairs using the Knuth shuffle [30]. Finally, we use the new sequence as the input for the search operation.

Our multi-threaded implementation is using OpenMP, an API for parallel computing based on the shared memory programming paradigm [14]. We also made use of PAPI to better understand the performance of our implementation. PAPI is an API for accessing available hardware counters inside the CPU [9].

6.2 CPU-optimized B$^+$-tree Evaluation

Memory Page Configuration. In this experiment, we aim to determine the memory page configuration that maximizes the search operation throughput. We evaluated our B$^+$-tree using three different configurations: (1) both I-segment and L-segment on small pages, (2) I-segment on huge pages and L-segment on small pages, (3) both I-segment and L-segment on huge pages.

To examine our expectation about the average TLB misses per query, we evaluated a single-threaded implementation of all three configurations and counted the TLB misses during search operations using PAPI. Since OpenMP library causes extra TLB misses, we excluded multi-threading to obtain more accurate measurement. We plot the average TLB miss per each query in Figure 7(a). Without utilizing huge pages, the misses increase as the tree grows. Also, it can be seen that searching in the implicit tree causes more TLB misses

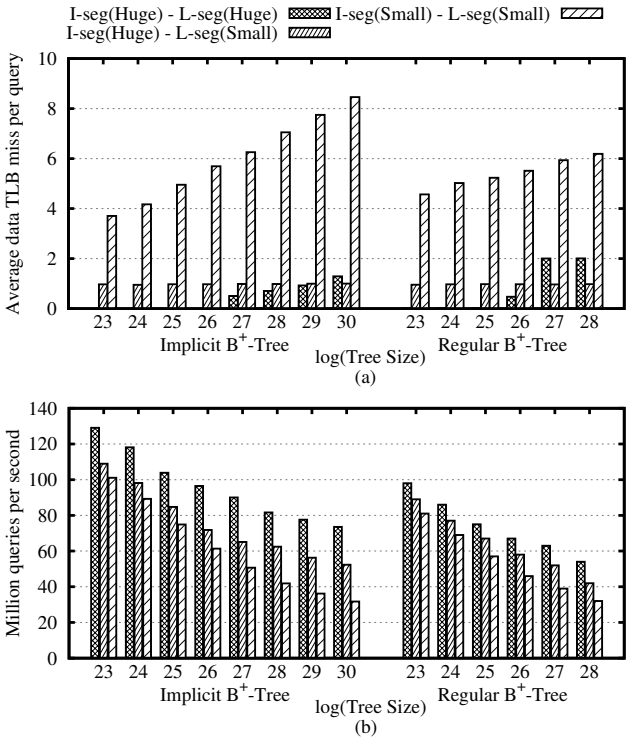


Figure 7: Memory page configuration evaluation. (a) TLB misses (b) Throughput.

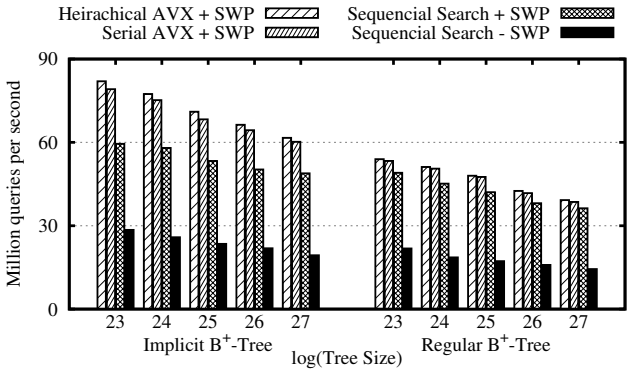


Figure 8: Software pipelining and node search comparison.

than for the regular B$^+$-tree. The reason is the fanout of inner nodes in the implicit tree is inferior to the regular tree, consequently, the tree depth is higher. Allocating only inner nodes on huge pages, significantly reduces the number of misses. In this case, misses are independent of tree depth and they are bounded to one TLB miss per query. Allocating the entire tree on huge pages eliminates misses for smaller trees which do not need more than 4GB of space. As the required space exceeds this amount, the average miss rate increases and surpasses one miss per query. We conclude from Figure 7(a) that in terms of TLB misses, the second configuration is more robust against tree growth, while the third one is best for trees less than 4GB in size.

To determine the effect of TLB misses on tree search performance, we evaluated the multi-threaded tree search using the same configurations. The results are in Figure 7(b). As expected, the first configuration is the least performing. The fastest configuration is the third one, although, it generates more TLB misses than the second configuration for bigger trees. According to our analysis, this behavior is the consequence of the different costs of misses for 4K and 1G pages. As a TLB miss occurs, a page walk is required to retrieve the requested physical address. For 4K pages, five memory accesses are required to translate logical to physical address, while three accesses are sufficient for 1G pages [25]. Even if the TLB miss rate is higher in the third configuration, the penalty of a page walk is less significant, which results in better performance. This experiment indicates the superiority of using huge pages in this application.

SIMD Accelerated Node Search. We now examine the node search algorithms to determine the fastest one and measure the resulting improvements. A query search operation is evaluated using three different search algorithms: (1) sequential, (2) linear SIMD, and (3) hierarchical SIMD; software pipelining is applied in all of these configurations. To indicate the effectiveness of software pipelining in this application, we also evaluated sequential search without software pipelining. Since AVX2 support is required for the evaluation, we evaluated this experiment on M_2 (M_1 does not support AVX2). The result of the experiments are given in Figure 8.

Enabling software pipelining is highly effective and improves the system throughout between 108%-152%, while it increased latency by 6X on average. Among the node search algorithms, the hierarchical SIMD approach, achieved the best result; it is slightly faster than linear search. Both SIMD implementations lose their advantage to sequential search as the tree size grows. This behavior confirms that tree processing becomes memory latency bounded for bigger trees, and memory optimization techniques are ever more important in this case.

Comparison with FAST. We compare our CPU-optimized implicit B$^+$-tree to FAST [29], the fastest reported indexing tree in the literature, – also an implicit structure, – to assure our CPU-optimized B$^+$-tree design is competitive enough to be used as a performance baseline. As shown in Figure 9, our B$^+$-tree achieved 1.3X higher throughput on average than FAST. Our different SIMD-enabled node search, which allows us to reach higher node fan-out and, consequently, better cache line utilization, is the source for this improvement. Even though our implementation achieves better performance than what FAST reported, we do not aim to challenge FAST in this work, since FAST is designed to be a configurable data structure, able to adapt to different hardware configurations, while our design is specifically tuned for the Intel architecture.

6.3 HB$^+$-tree Evaluation

Bucket Handling Strategies. In this experiment, we study three different bucket handling techniques: (1) sequential, (2) pipelining, and (3) pipelining with double buffering. With sequential bucket handling, it is neither feasible to employ CPU and GPU simultaneously, nor overlap communication and computation. This approach is the simplest; we use it as baseline in our evaluation. Resolving buckets using pipelining allows us to partially overlap CPU and GPU computations. Double buffering helps to overlap

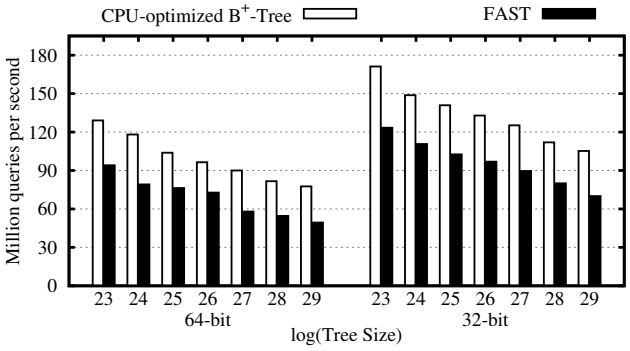


Figure 9: Comparison of FAST and implicit CPU-optimized B$^+$-tree.

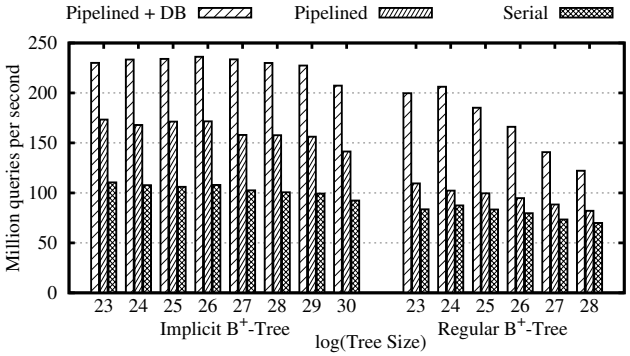


Figure 10: Bucket handling strategy evaluation.

data transfer and search, so that it improves resource utilization. We show the results for search using these techniques in Figure 10.

Sequential bucket handling is the least efficient. Pipelining is more effective for the implicit B$^+$-tree. It increases the throughput by 56% for implicit and by 20% for regular B$^+$-tree. The double buffering technique is effective for both tree versions. Using bucket pipelining extended by double buffering improves throughput by 110% over the baseline technique. Gaining twice the throughput in comparison to the sequential approach indicates that we successfully managed to simultaneously exploit the computation capabilities of both processors.

Bucket Size. The goal of this experiment is to determine the optimal bucket size considering both throughput and latency. Increasing the bucket size, diminishes the influence of communication and GPU initialization overheads, resulting in better system throughput, while at the same time, increases the system latency. We evaluated the search operation using M_1 for different bucket sizes: 8K, 16K, 32K, and 64K. As shown in Figure 11, search throughput grows, as bucket size increases for the implicit B$^+$-tree, while for the regular B$^+$-tree, the throughput is nearly the same for bucket sizes 16K, 32K, and 64K. Considering that the average latency also increases as the bucket size grows (2.7X for 64K and 1.7X for 32K), we use 16K as the optimal bucket size for the rest of our experiments.

Impact of Skewed Data. We studied HB$^+$-tree for several input data distributions, including Uniform, Normal($\mu = 0.5$, $\sigma^2 = 0.125$), Gamma($k = 3$, $\theta = 3$) and Zipf($\alpha = 2$). The generated random values are in the range

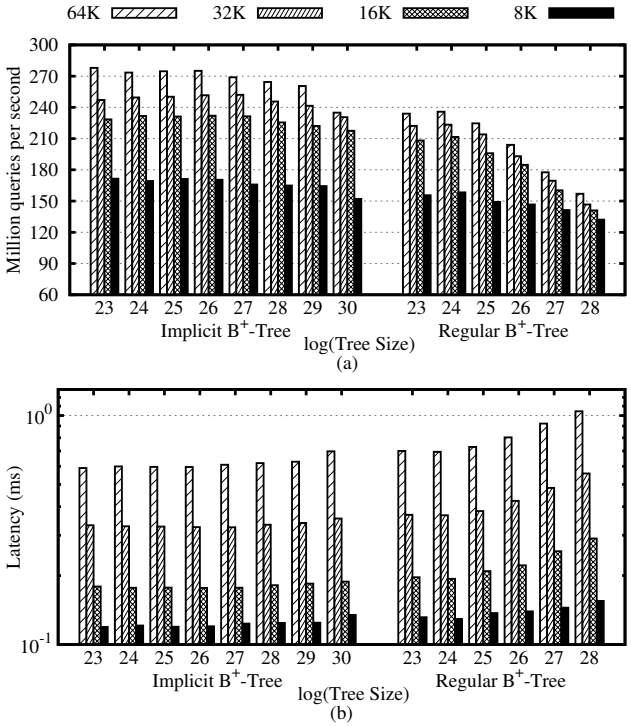


Figure 11: Experiment on varying size of buckets (a) throughput (b) latency.

$[0, 1]$. Before the values are given to search queries, they are linearly mapped to $[0, MAX]$. We used the Uniform distribution as the baseline and scaled the results of other distributions accordingly. The normalized results are illustrated in Figure 12.

The performance on all distributions, except Zipf, is within 1.1X of the Uniform distribution, while the performance for Zipf input data increases by up to 2.2X. When the data becomes more skewed, the same portion of the tree is accessed more frequently, which results in a higher cache hit rate. This behavior is even more pronounced for highly skewed data, such as the Zipf distribution.

Update Performance. In this experiment, we evaluate the performance of update query execution on HB$^+$-tree as compared to CPU-optimized B$^+$-tree for both regular and implicit tree versions.

We first present evaluations of the different update query execution methods for the regular HB$^+$-tree including both the single- and multi-threaded versions of the synchronous and asynchronous approach. Figure 13(a) illustrates the throughput of these methods for various tree sizes; the I-segment transfer time is excluded for the asynchronous approaches. Parallel execution is more effective in the asynchronous approach which results in 3X higher throughput in comparison with the single-threaded approach. The synchronous approach is only 30% faster than the multi-threaded one, which is bounded by the data transfer latency between CPU and GPU memory.

The I-segment synchronization times for different tree sizes are illustrated in Figure 13(b). To examine the effect of I-segment synchronization overhead, we measure the time required to perform batch updates with different batch sizes in a tree of size 64M. The results are shown in Figure 14.

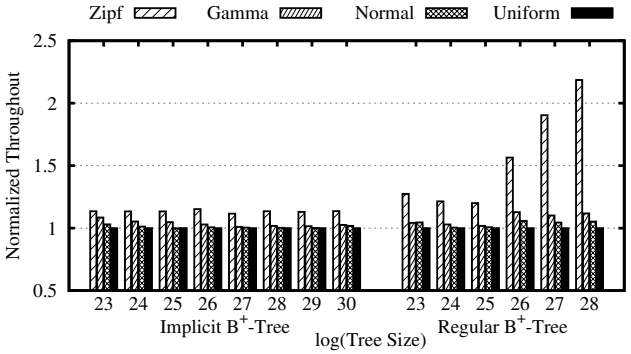


Figure 12: Experiment on different distributions.

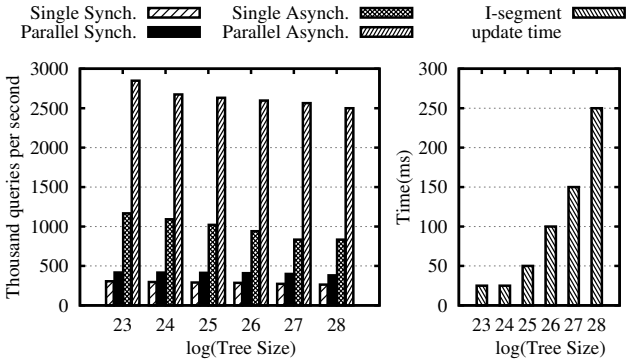


Figure 13: Evaluation of Regular B$^+$-tree update.

Up to a batch size of 64K, the synchronous approach performs better because of the slow I-segment transfer in the asynchronous approach. But for batches larger than 128K, the asynchronous is more effective, as the I-segment transfer cost is amortized by the larger number of queries processed. This experiment shows that the choice of update depends on the batch size. A synchronous update is more efficient for smaller batches while an asynchronous one performs better for larger batches.

To update implicit CPU-optimized B$^+$-tree, the entire tree has to be rebuilt, including the I-segment and L-segment. For implicit HB$^+$-tree, it is additionally required to transfer the I-segment to GPU memory. To compare the cost of updating these two trees, we measure the cost of each phase including L-segment rebuilding, I-segment rebuilding, and I-segment transfer separately as shown in Figure 15. The cost of transferring the I-segment is only 3 to 7 percent of tree reconstruction.

6.4 HB$^+$-tree vs. CPU-optimized B$^+$-tree

We now compare the search performance of HB$^+$-tree against the CPU-optimized B$^+$-tree in terms of throughput, latency and selectivity using M_1.

Throughput. Figures 16(a) and 16(b) show the search performance of both trees (for 64-bit and 32-bit variable sizes). The throughput of the implicit HB$^+$-tree is almost constant for different tree sizes, which indicates that the amount of time the GPU requires for traversing inner nodes is inferior to the time, the CPU requires for scheduling and searching leaf nodes. Consequently, the search performance is bounded by the computational power of the CPU. However, the regular HB$^+$-tree does not show similar behavior;

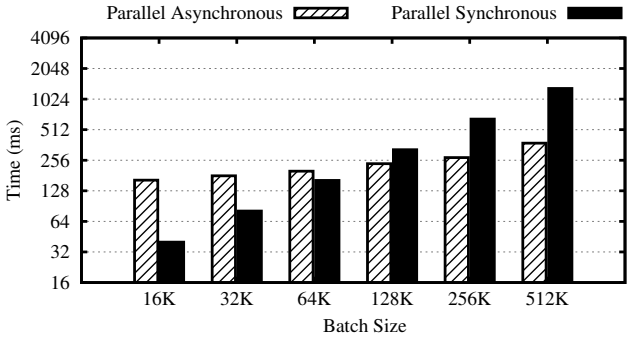


Figure 14: Regular B$^+$-tree update for different batch sizes.

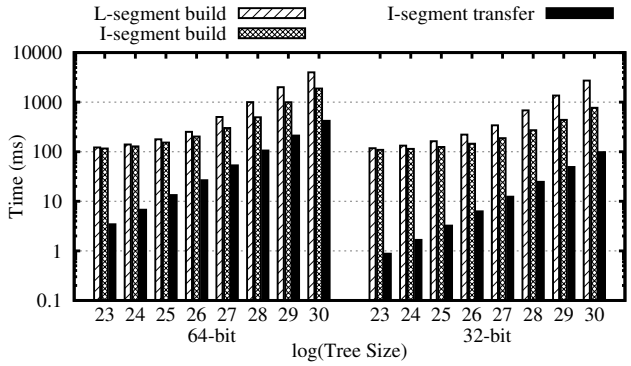


Figure 15: Implicit HB$^+$-tree update.

similar to the CPU-optimized tree, its performance declines as the tree grows. The GPU accelerated approach outperforms the CPU-optimized approach by 2.4X and 2.1X higher throughput on average for 64-bit and 32-bit variables, respectively.

Latency. Figure 16(c) illustrates the query search latency for both HB$^+$-tree and CPU-optimized B$^+$-tree. The hybrid approach exhibits comparably higher latency, 67X on average, than the CPU-optimized one. The higher latency is the consequence of a different number of queries required for an effective utilization of each platform. The number of concurrent queries for CPU and GPU are 2^8 and 2^{14}, respectively, where the ratio (64) is almost the same as the latency ratio. The average latency of the hybrid approach is less than 0.18ms for the implicit B$^+$-tree and 0.25ms for regular the B$^+$-tree.

Range queries. In this experiment, we compare HB$^+$-tree against CPU-optimized B$^+$-tree in performing range queries for different numbers of matching keys per query for a total of 128M keys. Figure 17 shows the performance of range queries for the retrieval of 1 to 32 keys. Since range queries require more leaf node traversal, the ratio of the search time in inner nodes to the entire lookup time decreases for these queries. As a result, the lookup performance of implicit and regular tree versions becomes similar; also, HB$^+$-tree looses its advantage as more keys per query match. HB$^+$-tree is more than 80% faster than the CPU-optimized B$^+$-tree up to 8 matching keys per query and the performance advantage decreases to 22% for 32 matching keys per query.

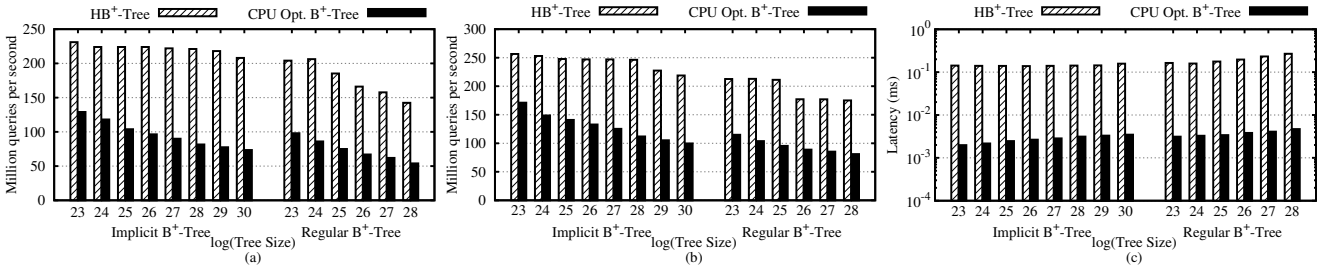


Figure 16: Evaluation of CPU-optimized B$^+$-tree and HB$^+$-tree. (a) Throughput (64-bit) (b) Throughput (32-bit) (c) Latency (64-bit).

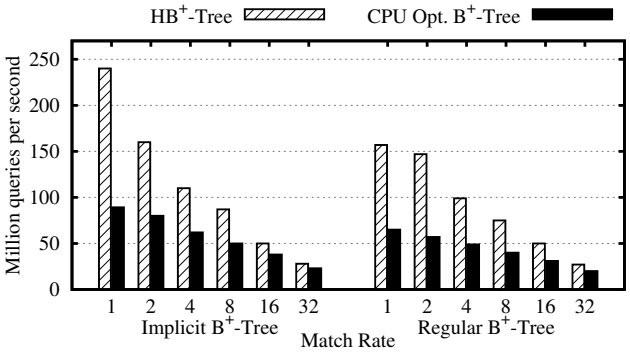


Figure 17: Throughput of Range queries.

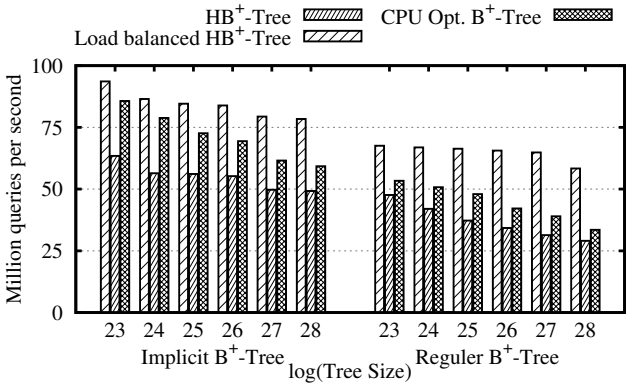


Figure 18: Evaluation of load balancing scheme.

6.5 Load Balancing Evaluation

We now examine the effectiveness of our load balancing scheme on heterogeneous platforms where the computation power is not bounded by the CPU. To this end, we used M_2 which is, relatively speaking, equipped with a less powerful GPU accelerator. Figure 18 shows the results. Without load balancing, HB$^+$-tree performs 25% slower than our CPU-optimized tree, on average. This indicates that the communication overhead between both processors is far higher than the acceleration provided by the GPU.

Applying load balancing scheme is highly effective and improves HB$^+$-tree throughput by 65% on average. In comparison to the CPU-optimized tree, the load balanced HB$^+$-tree performs up to 32% and 65% better for the implicit and regular approach, respectively.

7. CONCLUSIONS

In this paper, we presented an indexing structure, called HB$^+$-tree, specifically tailored to a heterogeneous computing platform with a hybrid memory architecture. Index search is accelerated by utilizing the resources of the hybrid GPU-CPU platform to aggregate the processing resources and memory bandwidth of both processing units. These improvements empower our approach to perform search faster for trees where the tree traversal performance approaches the memory bandwidth limit. In such situations, HB$^+$-tree performs on average 2.4X faster search than the CPU-optimized B$^+$-tree, with individual measurements improving performance by up to 2.9X.

The directions for our future work are two-fold: (1) Further support for parallel update queries and (2) development of a general leaf-stored tree processing framework using a CPU-GPU hybrid platform. In this paper, we primarily focused on realizing efficient search. So far, updates are performed sequentially by the CPU with asynchronous data transfer to the GPU; this could be further improved by employing GPU cycles in support of parallel update query execution. The other direction is to develop a general framework which enables the use of a CPU-GPU hybrid platform for any arbitrary leaf-stored tree structure, such that using the node structure and search/update function as input, the framework would determine the parameters for an approach that best utilizes the resources of both CPU and GPU.

8. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2009.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, 2012.

[3] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *ICDE*, 2015.

[4] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[5] M. Athanassoulis and A. Ailamaki. Bf-tree: Approximate tree indexing. *Proc. VLDB Endow.*, 2014.

[6] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, 2010.

[7] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *SC*, 2011.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, 1970.

[9] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *IJHPCA*, 2000.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD*, 1997.

[11] S. Chen, P. B. Gibbons, and T. C. Mowry. *Improving index performance through prefetching*. ACM, 2001.

[12] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[13] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an APU. In *SCC*, 2012.

[14] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 1998.

[15] R. Elmasri. *Fundamentals of database systems*. Pearson Education India, 2008.

[16] J. Fix, A. Wilkes, and K. Skadron. Accelerating braided b+ tree searches on a GPU with CUDA. In *A4MMC*, 2011.

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[18] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 2011.

[19] M. Grand. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*, volume 1. John Wiley & Sons, 2003.

[20] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, 2011.

[21] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. In *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[22] B. C. O. K.-L. T. M. Z. Hao Zhang, Gang Chen. In-memory big data management and processing: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 2015.

[23] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, 2008.

[24] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[25] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-052US. September 2014.

[26] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of data warehouses*. Springer Science & Business Media, 2013.

[27] K. Kaczmarski. B+-tree optimized for GPGPU. In *OTM*. 2012.

[28] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. In *HotPar*, 2009.

[29] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.

[30] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, The*. Addison-Wesley Professional, 2014.

[31] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, 1986.

[32] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[33] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

[34] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 2010.

[35] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*. 2000.

[36] H. Lu, Y. Y. Ng, and Z. Tian. T-tree or b-tree: Main memory database index structure revisited. In *ADC*, 2000.

[37] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[38] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 1980.

[39] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE micro*, 2010.

[40] C. Nvidia. NVIDIA CUDA programming guide (version 6.5). *NVIDIA Corporation*, 2014.

[41] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. 2008.

[42] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. VLDB, 1999.

[43] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, 2000.

[44] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[45] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment*, 2011.

[46] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[47] A. Vaisman and E. Zimányi. Data warehouses: Next challenges. In *Business Intelligence*. Springer, 2012.

[48] P. Vassiliadis and A. Simitsis. Near real time ETL. In *New trends in data warehousing and data analysis*. Springer, 2009.

[49] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH*, 1995.

[50] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.

APPENDIX

Here, we provide more details on our implementations of the SIMD-enabled node search using AVX unit, the GPU search kernel, and more evaluations.

A. SIMD ENABLED SEARCH

In this section, we present our SIMD enabled search algorithm in more detail. Considering `Node[0..7]` is an array of keys in an inner node and `query` is the given search query, Snippets 1 and 2 show the implementation of the linear and hierarchical approaches for 64-bit keys, respectively.

Snippet 1 Linear AVX search (64-bit)[§].

```
1:  __m256i†Vquery =_mm_set1_epi64x(query);
2:  __m256i vec =_mm256_set_epi64x(node[0],
3:  node[1], node[2], node[3]);
4:  __m256i Vcmp =_mm256_cmpgt_epi64(Vquery,
        vec);
5:  int cmp = _mm256_movemask_epi8(Vcmp);
6:  cmp = cmp & x10101010;
7:  cmp = __builtin_popcount‡(cmp);
8:  int k = cmp;
9:  Vec = _mm256_set_epi64x(node[4], node[5],
10: node[6], node[7]);
11: Vcmp = _mm256_cmpgt_epi64(Vquery, vec);
12: cmp = _mm256_movemask_epi8(Vcmp);
13: cmp = cmp & x10101010;
14: cmp = __builtin_popcount (cmp);
15: k += cmp;
16: // k is the minimum i s.t. query <= node[i]
```

[§]Functions starting with `_mm` are SIMD instructions

[†]256-bit data type as four 64-bit integer values

[‡]method by GNU's Compiler Collection (GCC) determines the number of ones in the binary representation of a number

Snippet 2 Hierarchical AVX search (64-bit).

```
1:  __m128i†Vquery = _mm_set1_epi64x(query);
2:  __m128i Vec =_mm_set_epi64x(node[2], node
        [5]);
3:  __m128i Vcmp =_mm_cmpgt_epi64(Vquery, Vec);
4:  int cmp = _mm_movemask_epi8(cmpRes);
5:  cmp = cmp & 0x00001010;
6:  cmp = __builtin_popcount (cmp);
7:  int k = cmp * 3;
8:  Vec = _mm_set_epi64x(node[k], node[k + 1]);
9:  Vcmp = _mm_cmpgt_epi64(Vquery, Vec);
10: cmp = _mm_movemask_epi8(Vcmp);
11: cmp = cmp & 0x00001010;
12: cmp = __builtin_popcount (cmp);
13: k += cmp;
14: // k is the minimum i s.t. query <= node[i]
```

[†]128-bit data type as two 64-bit integer values

The linear approach first loads the `query` into an AVX vector in Line 1. In Lines 2-4, the first half of the key array (`Node[0..3]`) is loaded into a vector and compared to `key`. Then, the number of keys smaller or equal to the input are stored in variable `k` (cf. Lines 5-8). This process repeats for the second half of the key array adding the comparison result to `k` in Lines 9-15. At the end, `k` is the index of the child to resolve the query.

The hierarchical approach first compares the boundary keys which are `node[2]` and `node[5]` to `query` in Lines 2-4 and based on the comparison results, the search algorithm calculates the index of keys for the second comparison and stores it in `k`. Finally, it compares the `query` to the `node[k]` and `node[k+1]` to find the right child index.

B. ADDITIONAL EVALUATIONS

In this section, we provide further experimental results for our HB$^+$-tree and our CPU-optimized B$^+$-tree.

B.1 HB$^+$-tree lookup using CPU

Figure 19 shows a comparison of the lookup performance of CPU-optimized B$^+$-tree and HB$^+$-tree only using the CPU. The performance of the regular tree versions are identical since they are based on the same node structures. The CPU-optimized implicit B$^+$-tree results in better performance, due to better cache line data utilization. The fanout of inner nodes in HB$^+$-tree is decremented by one for the benefit of faster search with GPU.

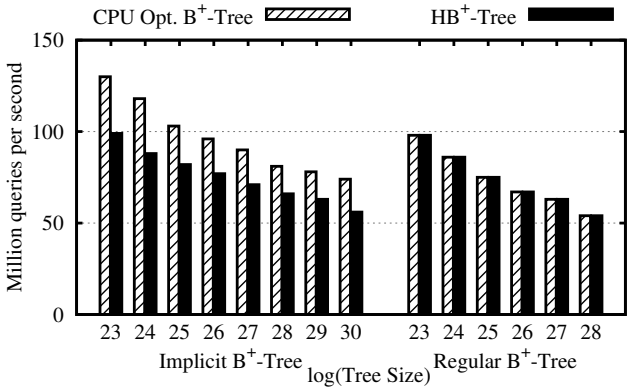


Figure 19: Evaluation of lookup in HB$^+$-tree using CPU.

B.2 Software Pipelining

In this experiment, we study the effect of software pipelining on lookup performance, that is, on throughput and latency. Software pipelining helps the processor to overlap execution with data fetching by executing multiple queries simultaneously, but at the same time, increases processing latency. Algorithm 2 shows how we apply software pipelining with prefetching for CPU-optimized B$^+$-tree lookup. Here, as a thread finishes searching a node, instead of waiting for the child node to be loaded into the cache, it switches to processing another query. Then, when the thread switches back to the same query, the child node is already loaded into cache.

Figure 20(a) illustrates the lookup throughput for various numbers of simultaneously processed queries ranging from 1 to 32. Increasing the number of queries from 1 to 16 continuously improves the throughput, which results in 2.5X better performance than without software pipelining. But due to the limited cache size, increasing the number of simultaneously processed queries from 16 to 32 is not effective and performance remains almost the same. The lookup latency for different software pipeline lengths is illustrated in Figure 20(b) which indicates the latency is quickly increasing with number of queries per thread. On average, the lookup

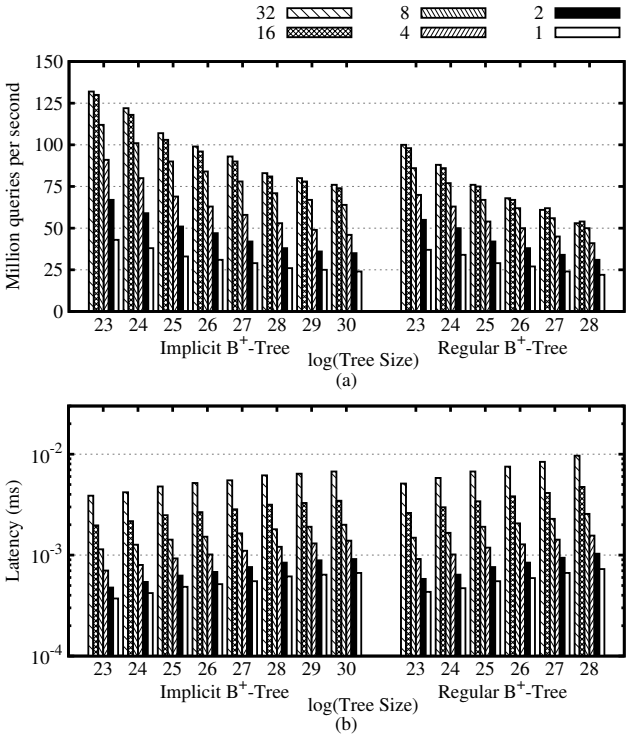


Figure 20: Evaluation of various software pipelining lengths: (a) throughput (b) latency.

latency using a software pipeline of length 16 is 6X higher than without software pipelining.

Algorithm 2 Software pipelining-enabled tree search

Input: P : length of software pipeline
Input: $keys[P]$: search queries
Input: H : height of B$^+$-tree

```
1: for i ← 1 to P do
2:     node[i] ← root
3: for step ← 1 to H − 1 do
4:     for i ← 1 to P do
5:         node[i] ← getNextNode(I-seg, node[i], keys[i])
6:         prefetch(node[i])
7: for i ← 1 to P do
8:     value[i] ← getValue(L-seg, node[i], key[i])
```

B.3 Concurrent Search and Update Queries

We now examine the performance of parallel search/update query execution in HB$^+$-tree utilizing only the CPU. We evaluate both synchronous and asynchronous approaches, where the I-segment transfer time is excluded for the asynchronous approach.

The synchronous approach consists of one synchronizing thread, which continuously updates the inner nodes in GPU memory and multiple query processing threads, while the asynchronous approach only consists of query processing threads. The query processing threads are based on the update algorithms given in Section 5.6, which are also capable of resolving search queries. The results are shown in Figure 21. As the ratio of update queries increases, the throughput of the synchronous approach decreases faster than the asynchronous one which is due to the high communication initialization overhead between GPU and main memory. The execution of buckets with 100% search queries in this evaluation is not as fast as our previously evaluated lookup methods which is due to the mutex locking and synchronization overhead in the query processing threads.

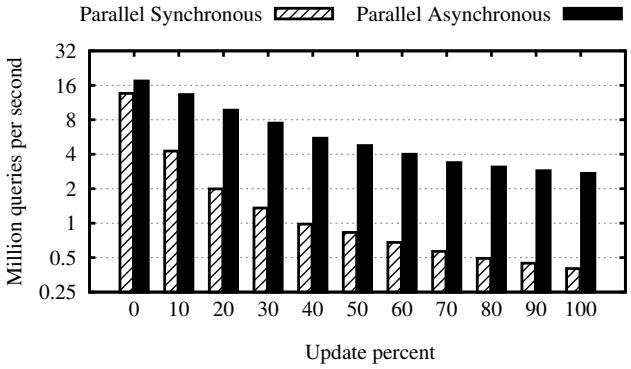


Figure 21: Evaluation of concurrent search/update queries.

C. CUDA PROGRAMMING MODEL

The CUDA programming model is based on hierarchical threading including the notions of *grid*, *block* and *thread*. At the top of the hierarchy is the grid which consists of *blocks* and each block is a group of threads. Threads are distinguished by two index values, `BlockIdx` and `ThreadIdx`, which define the runtime behavior of each thread.

Implementing efficient programs for CUDA requires good understanding of both the GPU memory architecture and the thread scheduling model. The unit of scheduling in CUDA is the *warp*, which is a set of 32 threads. Threads within the same warp are able to execute only a single instruction at a time. The situation called *warp divergence* results when threads of the same warp divert into different code paths, typically occurring for if-then-else statements. As a result, the warp has to be scheduled for both paths separately, which increases the total execution time. Proper index assignment, which directs threads of each warp into a single code path avoids this situation.

Since there is no message passing mechanism supported by CUDA, communication is only feasible via read/write operations on memory. Therefore, efficient bandwidth utilization is of great significance. The CUDA memory architecture is composed of different memory types, where the important ones for our work are *shared memory* and *device memory*.

Device memory, or GPU memory, is the biggest and slowest memory; it can be accessed by an entire grid. All accesses to device memory have to be done through either 32-, 64-, or 128-byte memory transactions. As a warp issues a device memory access, the GPU coalesces this access into transactions of these sizes. In the worst case, each access is translated into 32 separate memory transactions, which divide the device memory performance by 32. Relocating data, such that threads within a warp access adjacent memory locations, leads to more efficient device memory utilization. Shared memory is comparably faster than device memory but it is limited to a block. To provide higher

Snippet 3 GPU kernel code for searching inner nodes in implicit tree ($F_I = 8$)[§].

```
1:  // teamQuery is the requested key
2:  long teamQuery;
3:  __shared__† char flag[9], result;
4:  char selfFlag, i;
5:  long selfKey, nodeIndex;
6:  flag[threadIdx.x] = 0;
7:  nodeIndex = 0; // root index
8:  __syncthreads()‡;
9:  for (i = 0; i < tree_depth; i++) {
10: // levelOffsets is an array stores the
11: // offsets of each level in tree
12: selfKey = tree[levelOffsets[i] + nodeIndex
        + threadIdx.x];
13: flag[threadIdx.x+1] = 0;
14: selfFlag = 0;
15: if (teamQuery <= selfKey) {
16: flag[threadIdx.y][threadIdx.x+1] = 1;
17: selfFlag = 1;
18: }
19: __syncthreads();
20: if (selfFlag == 1 &&
21: flag[threadIdx.x] == 0) {
22: result = threadIdx.x;
23: }
24: __syncthreads();
25: // threads traverse to the next node
26: nodeIndex = (nodeIndex + result)<<3;
27: }
28: // nodeIndex is the index of target leaf
        node
```

[§]This program is executed by eight threads concurrently with `threadIdx.x` = 0 to 7

[†]Shared variables are declared by `__shared__`

[‡]`__syncthreads` is a barrier synchronization primitive

bandwidth, shared memory is composed of multiple memory banks which can be accessed simultaneously. The highest bandwidth is achieved when accesses are equally separated among the memory banks. For inter-block communication, shared memory is the better option in comparison to the device memory due to lower access latency.

The Nvidia Tesla platform is explicitly targeting high performance computing and offers faster double precision floating point operations which is a critical requirement for many applications. However, we opted for the Nvidia Geforce processor family in this work, since index tree search algorithms only require integer operations, either 64-bit or 32-bit, and the Geforece platform offers a better computation-power-to-price ratio for these operations.

D. GPU SEARCH KERNEL

The Snippet 3 represents the GPU kernel function for searching the I-segment of the implicit HB$^+$-tree. The input parameters are `I-seg`: the reference to the I-segment in GPU memory, `levelOffsets`: offsets of each level in I-segment, and `teamQuery`: the given search query. Shared variables are declared by the keyword `__shared__` and barriers `__syncthreads` are used to avoid race conditions when accessing shared variables.

Search starts from the root node (`nodeIndex = 0`) and performs the parallel node search per each inner node. Each thread loads a key from the current node and stores it into local register (`selfKey`) in Line 12. After initializing flags, threads compare their local register to `teamQuery` and store the result in both local and shared flags. In Line 19, it is required to synchronize threads before they check the shared flag to avoid race conditions. In Lines 20-23, each thread deduces if it is assigned to the next level node. If so, the thread update the shared result variable. The operation is repeated for each level of inner nodes. At the end, `nodeIndex` is referring to the desired leaf node.

Appendix B

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,

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380576

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.

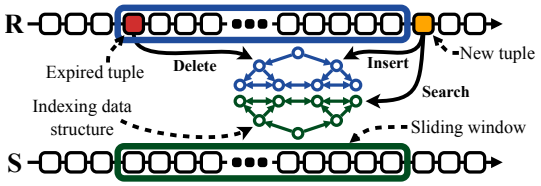


Figure 1: Index-based window join.

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

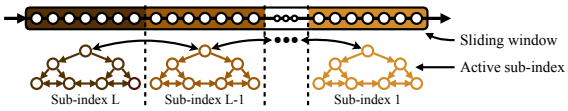


Figure 2: Chained index.

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

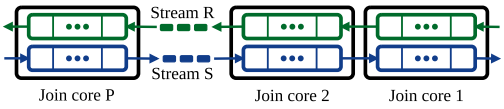


Figure 3: Low-latency handshake join.

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

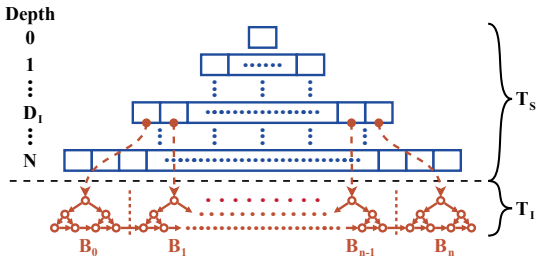


Figure 4: Structure of PIM-Tree (blue and red sections are T_S and T_I components, respectively).

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.

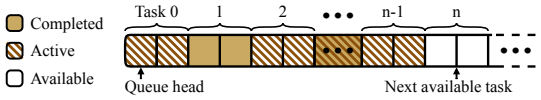


Figure 5: Shared task queue with task size of 2.

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

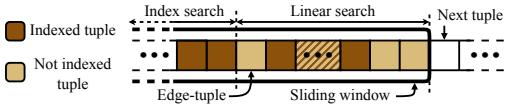


Figure 6: Sliding window during parallel stream join.

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

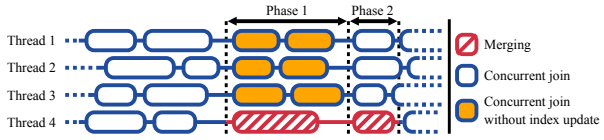


Figure 7: Nonblocking merge.

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].

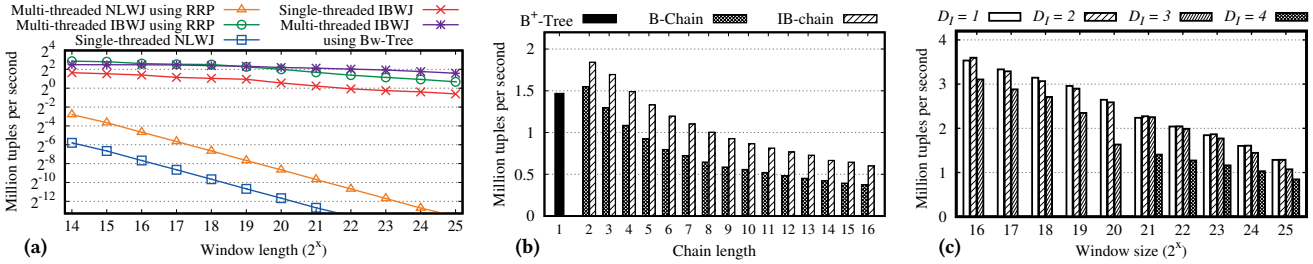


Figure 8: a) Performance evaluation of multithreaded window join using round-robin partitioning (RRP). b) Throughput comparison of IBWJ using chained index and B$^+$-Tree. c) Throughput vs. insertion depth for single-threaded IBWJ using PIM-Tree.

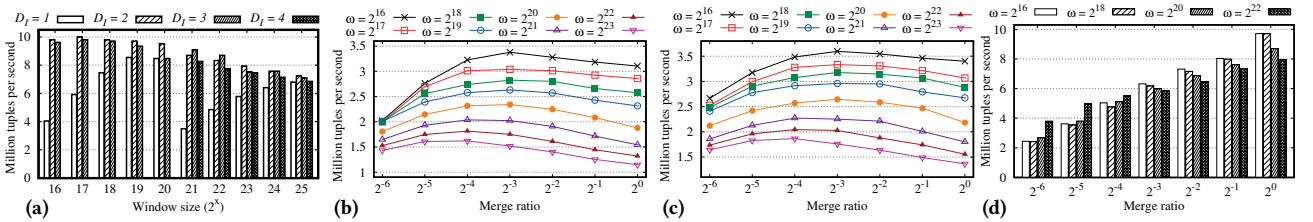


Figure 9: a) Throughput vs. insertion depth for parallel IBWJ using PIM-Tree. b) Throughput vs. merge ratio for IBWJ using IM-Tree. c) Throughput vs. merge ratio for IBWJ using PIM-Tree. d) Throughput vs. merge ratio for parallel IBWJ.

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.

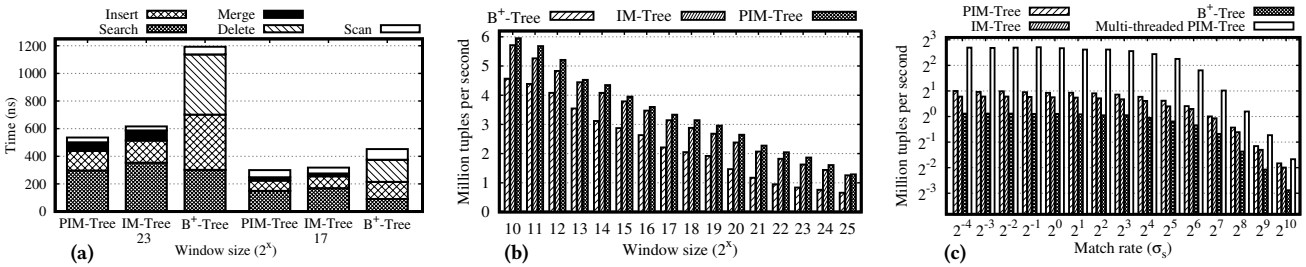


Figure 10: a) Cost comparison of the different steps of IBWJ for a single tuple using various indexing data structures. b) Performance comparison of single-threaded IBWJ using different indexing data structures. c) Throughput vs. match rate for IBWJ.

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

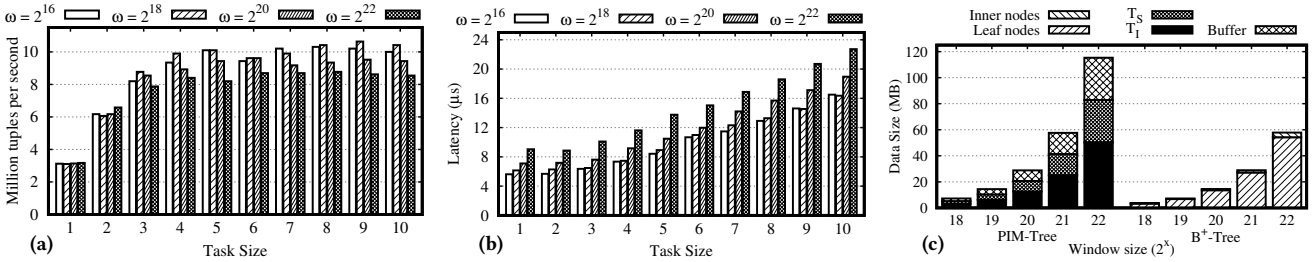


Figure 11: a) Throughput vs. task size for parallel IBWJ using PIM-Tree. b) Latency vs. task size for parallel IBWJ using PIM-Tree. c) Memory footprint comparison of B⁺-Tree and PIM-Tree.

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

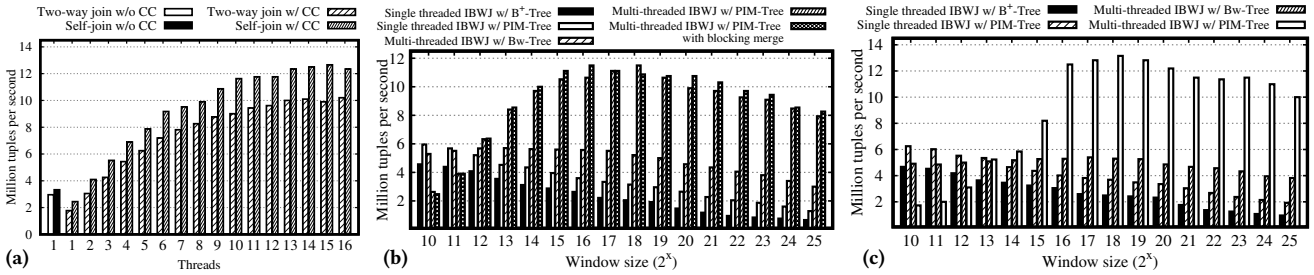


Figure 12: a) Comparison of parallel IBWJ using PIM-Tree utilizing varying number of threads against the single-threaded implementation without concurrency control (CC) ($w = 2^{20}$). b) Throughput comparison of single-threaded and multithreaded two-way join. c) Performance comparison of single-threaded 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, $\theta = 3$ and k = 1, $\theta = 5$), 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

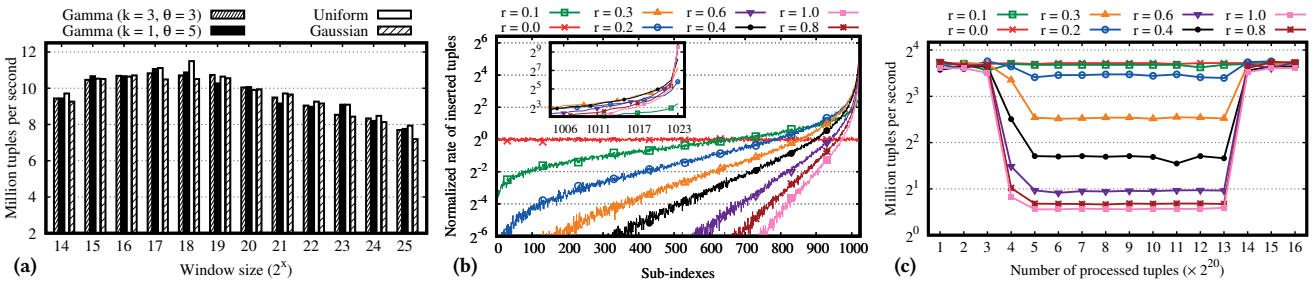


Figure 13: a) Evaluation of parallel IBWJ using PIM-Tree for different tuple value distributions. b) Distribution of inserts among subindexes during drifting Gaussian distributions. c) Evaluation of multithreaded index-based self-join using PIM-Tree for shifting Gaussian distributions.

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 multithreaded 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.

Appendix C

Distributed Stream KNN Join

Amirhesam Shahvarani
Technische Universität München
Munich, Germany
ah.shahvarani@tum.de

Hans-Arno Jacobsen
University of Toronto
Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

*k*NN join over data streams is an important operation for location-aware systems, which correlates events from different sources based on their occurrence locations. Combining the complexity of *k*NN join and the dynamicity of data streams, *k*NN join in streaming environments is a computationally intensive operator, and its performance can be greatly improved by utilizing the computational capabilities of modern non-uniform memory access (NUMA) computing platforms. However, the conventional approaches to *k*NN join for prestored datasets do not work efficiently with the kind of highly dynamic data found in streaming environments.

Therefore, in this paper, we introduce an adaptive scalable stream *k*NN join, named ADS-kNN, to address the challenges of performing the *k*NN join operation on highly dynamic data. We propose a multistage *k*NN execution plan that enables high-performance *k*NN queries in distributed settings by overlapping the computation and communication stages. Moreover, we propose an adaptive data partitioning scheme that dynamically adjusts the load among the operators according to the changes in the input values. Combining these two techniques, ADS-kNN provides a scalable and adaptive *k*NN join operator for data streams. Our experiments using a 56-core system show that ADS-kNN achieves a maximum throughput that is 21 times higher than that of a single-threaded approach.

CCS CONCEPTS

• **Information systems** → **Data streams**; *Parallel and distributed DBMSs.*

KEYWORDS

Distributed computing; Data streams; Nearest neighbor join

ACM Reference Format:
Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed Stream KNN Join. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457269

1 INTRODUCTION

With advances in wireless communication and mobile devices, location-aware services have become increasingly popular in the past decade [7, 26, 37, 40]. Many location-aware applications, such as social networking platforms, recommender systems and location-based games, gather excessive amounts of geospatially tagged streaming data at an ever-increasing pace [18, 23, 38]. A high performance data processing solution is a crucial requirement in these applications to provide real-time service [13, 28, 29, 31, 32].

The *k* nearest neighbor (*k*NN) join is a computationally intensive and frequently used operation that is supported by many location-aware services [8, 33]. Given the two datasets R and S, *k*NN join correlates each tuple from *R* to its *k* nearest tuples in *S*. Stream *k*NN join is a special type of *k*NN join, in which the two datasets *R* and *S* are given in the form of data streams. *k*NN join is a useful operation in many streaming scenarios, such as locating the taxis nearest the customer's location, correlating a tweet with the geospatially nearest tweets, and finding the nearest photos to a user in a photo-sharing platform [14, 30]. Combining the complexities of the join operation with *k*NN query processing and the dynamicity of streaming data, the stream *k*NN join is a computationally intensive operator, and single-threaded approaches cannot meet the desired performance in many scenarios. Therefore, a scalable multithreaded solution that is capable of exploiting the computational power of multiple processors is desirable. Although there are different approaches for distributed *k*NN join over static (i.e., pre-stored) datasets, they are not applicable in a streaming setting. Data preprocessing is a common and effective technique for partitioning a static dataset. However, preprocessing is not an option in data streams. Data are more dynamic in a streaming environment than in conventional databases, where data are updated through ETL processes. Therefore, the join mechanism is required to continuously adapt to the changes in the data distribution in real time.

Depending on their underlying hardware, parallel stream processing systems can be grouped into two categories: single-node (scale-up optimized) or multinode (scale-out optimized). Scale-up optimized systems, such as Trill [5] and StreamBox [22], are focused on effective algorithms to utilize the resources of a single-node workstation, which are often based on a *non-uniform memory access* (NUMA) architecture. Compared with multinode systems, scale-up optimized stream processing systems employ more sophisticated task distribution and communication techniques to exploit the communication network among processors. In contrast, scale-out optimized systems, such as Flink [4], Storm [36] and Spark [41], are designed for stream processing on a multinode computing server. These systems are often based on massive data parallelism to exploit computational resources, and tasks are distributed among processors using a producer-consumer pattern. As a consequence, these systems achieve suboptimal performance using a single-node computing machine in comparison with scale-up optimized stream processing systems or multinode systems optimized for both scale-up execution and scale-out execution, such as IBM

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457269

System S [11, 15]. With advances in modern NUMA computers in terms of both memory storage and computational power, single-node stream processing solutions based on scale-up optimization are sufficiently powerful for many applications and have become an interesting alternative for scale-out optimized systems [9, 42].

Thus, in this paper, we introduce *adaptive distributed stream kNN join (ADS-kNN)* as a solution for scalable and real-time kNN join for streaming data. Unlike existing approaches that are based on data preprocessing, load balancing in ADS-kNN is based on online data distribution analysis and repartitioning. Therefore, ADS-kNN can be applied in any real-time streaming system, and it does not require any prior knowledge about the input data distribution. Moreover, we introduce a multistage query execution mechanism that enables ADS-kNN to overlap data communication and query execution.

A scalable parallel operator requires an effective data partitioning mechanism that simultaneously considers both workload balance and scalability. Simple partitioning mechanisms, such as round-robin, provide a uniform workload among operators, but these mechanisms are not scalable approaches for the problem of distributed *k*NN join. When using round-robin partitioning, all partitions must be queried to determine the *k*NN of a single tuple, as tuples are assigned to partitions independently of their values. Therefore, the cost of *k*NN join using round-robin increases almost linearly with an increasing number of join operators in the system, which limits its scalability. Hash partitioning is also not effective for *k*NN queries because hash functions do not preserve the spatial distances of tuples, and for this reason, we require all partitions to be queried for every *k*NN query. Therefore, we opt to use a spatial data structure as the partitioning function in our system. Using space partitioning, we can limit the scope of each *k*NN query to a subset of join operators, and therefore, maintain the scalability of our system. However, proposing a space partitioning method that provides a balanced workload while adapting to the data distribution changes is a challenging endeavor.

We propose an adaptive space partitioning mechanism that is based on two separate processes: workload monitoring and partition updating. ADS-kNN constantly evaluates the effectiveness of the current data partitioning based on periodic load statistics submitted by the join operators. If the system determines that the current data partitioning is not effective, it generates a new data partitioning using an approximate data distribution gathered from the join operators and initiates a data repartitioning process. Regarding the partitioning update, we propose two partitioning update methods, *lazy* and *instant*, which are useful for different scenarios. The instant partitioning update is a blocking operation, but it maintains the *k*NN-join operation in an efficient state. The lazy partitioning update is a nonblocking operation, but more time is needed for the system performance to recover using this method than the instant partitioning update. Furthermore, as a complement to our data partitioning approach, we propose a concurrent *k*NN query execution mechanism that enables low-latency result delivery and efficient utilization of the available computational resources. This multistage design enables our system to overlap the communication and computation stages of *k*NN query to achieve better resource utilization.

To validate the ideas developed in this paper, we design and develop a lightweight stream processing framework based on MPI [21], a message passing interface designed for parallel computing. Although we use it only for the distributed *k*NN join in this paper, our stream processing framework is generic and capable of performing other types of streaming operations. Distributed data processing systems are mostly built based on two distinct stages: data distribution and query execution. It is important to dedicate an adequate ratio of computational resources to each stage to maximize the system utilization. The basic solution to this problem is to empirically determine the optimal ratio using benchmarks for each query and system configuration. However, this method has two shortcomings. First, this method requires performing a benchmark before each query execution; second, this technique is not effective for scenarios where the optimal ratio is dynamic and changes during query runtime. To address this issue, we propose a unified operator model that enables dynamic load balancing between different stages of the join process. Utilizing 52 join operators, our distributed stream *k*NN join using ADS-kNN results in approximately 30 times higher throughput than the single-join-operator implementation. This result indicates a parallelization efficiency of 57%. Moreover, ADS-kNN outperforms a distributed stream *k*NN join using round-robin partitioning, which is employed in multiple state-of-the-art stream join approaches, by a factor of 12.

In summary, the contributions of this paper are listed as follows: (1) We propose a scalable multistage *k*NN query execution method to achieve a high-performance and low-latency distributed *k*NN join. (2) We propose an adaptive data partitioning method that adjusts the load distribution according to the input data streams. (3) We design and develop a lightweight stream processing framework to implement the ideas presented in this paper. (4) We develop an analytical comparison between our approach and state-of-the-art methods to provide better insight into our design decisions. (5) We conduct an extensive experimental study of ADS-kNN and provide a quantitative comparison with a state-of-the-art approach.

The remainder of this paper is organized as follows: In Section 2, we review the related approaches. In Section 3, we define the problem of a distributed *k*NN join on data streams. In Section 4, we describe the ADS-kNN architecture and how to perform the *k*NN join. Section 5 describes the implementation details of our distributed stream join system. Section 6 provides the cost analysis of ADS-kNN. We present our experimental results in Section 7. Section 8 concludes our work and defines our future research directions.

2 RELATED WORK

Work related to our approach can be classified into three categories: distributed kNN processing, stream kNN processing, and the join operation over streams.

Distributed kNN processing. Because of its computational complexity and importance in many applications, distributed *k*NN processing has received considerable attention in recent years. Hadoop-GIS [1] is a spatial data warehousing system that is based on the MapReduce model. Hadoop-GIS is integrated into Hive and provides a spatial query language extension to enable parallel spatial queries through MapReduce tasks. MD-HBase [26] proposes a multidimensional index layer over HBase for efficient range and kNN queries. Using a linearization technique, this method transforms multidimensional data points into a single-dimensional space and stores them in a range-partitioned key-value store. Likewise,

SpatialHadoop [10] extends Hadoop to enable native support for spatial query execution, and SparkGIS [3] is an Apache Spark extension for spatial data processing. However, all previously mentioned works focus on the analysis of prestored datasets, and their approaches do not apply to streaming environments.

Adaptive query workload aware (AQWA) [2] is a data partitioning mechanism for spatial data based on a kd-tree. AQWA performs data partitioning at initialization and repartitions data for each input batch. Although AQWA proposes an adaptive partitioning solution, this solution is based on preprocessing data and does not apply to real-time data processing streaming applications. Chatzimilioudis et al. [6] introduced Spitfire, a solution for kNN self-join over a set of objects utilizing a shared-nothing distributed system. Likewise, Spitfire is based on batch preprocessing, which means that it is not applicable to streaming data, which is the central focus of our work.

Stream kNN processing. There are a group of works that are also focused on processing kNN queries on streaming data. Koudas et al. [17] introduced an approximate kNN on a sliding window based on adaptive indexing using space-filling carves, which finds the k nearest neighbors within a given error bound. Mouratidis et al. [24] proposed a solution for continuous nearest neighbor queries on data streams based on conceptual partitioning and reducing nearest neighbor monitoring to the skyline maintenance problem to reduce the computation costs. Yang et al. [39] proposed a high-dimensional R-tree (HDR-tree) to address the reverse kNN problem on data streams. However, the scope of all these efforts is limited to single-threaded solutions. Pripuzic et al. [24] proposed a distributed kNN processing system for publish/subscribe systems based on a content-addressable network overlay. Their solution is based on static domain partitioning and does not apply to scenarios where the data distribution changes.

Join operation over streams. Another category of work related to our approach is stream join. Golab et al. [12] evaluated different sliding-window indexing approaches, such as hash-based and tree-based indexing, for different types of stream join operators. Kang et al. [16] evaluated the performance of an asymmetric sliding stream join using different algorithms, such as the nested loop join, hash-based join, and index-based join. Both approaches consider only a single-threaded stream join.

Several approaches have explored parallel stream join operators. Handshake join is a scalable NUMA-aware stream join that propagates stream tuples along a linear chain of cores in opposing directions [34]. Roy et al. [27] enhanced 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 storing and processing steps to reduce the dependency among processing units [25]. Lin et al. [19] proposed a real-time and scalable join model for a computing cluster by organizing processing units into a bipartite graph to reduce the memory requirements and dependency among processing units. All these approaches considered only a range join over data streams, while the focus of our work is kNN queries. Moreover, unlike our method, which is based on space partitioning to distribute data among operators, data partitioning in all the previously mentioned approaches is performed based on a round-robin order. In our evaluation, we present the performance of distributed kNN queries using round-robin partitioning and compare it to our space partitioning-based solution.

3 PROBLEM DEFINITION

This section provides an introduction to the stream kNN join and space partitioning. The notation we use throughout this paper is given here.

$\mathcal{D}$	:	A d-dimensional space
$a.t$	:	Arrival time of tuple a
$a.p$	:	Location of tuple a in space
$\|p, q\|$	:	Distance between two points p and q
$W_A(t)$	:	Sliding window of stream A at time t

The two streams R and S are input into the system, and each tuple t from either stream is associated with a point $t.p$ that represents the position of the tuple t in the d-dimensional space D.

3.1 Stream kNN Join

***k*NN search.** Given the tuple t and the tuple set S, the kNN of t over S is denoted as $kNN(t, S)$ and defined as follows: $kNN(t, S)$ is a subset of S with a maximum size of k, where the distance of all other elements of S to the tuple t is greater than or equal to the distance of each element of $kNN(t, S)$ to the tuple t.

$$kNN(t,S) \subset S \qquad |kNN(t,S)| = min(k, |S|)$$

$$\forall s_i \in kNN(d,S), s_j \in S - kNN(d,S) \rightarrow |t.p, s_i.p| \leq |t.p, s_j.p|$$

***k*NN window join.** Given two streams R and S, the kNN of R over S is denoted as $R \ltimes_{kNN} S$. For every incoming tuple r from R, $R \ltimes_{kNN} S$ generates a tuple of the form $(r, kNN(r, W_S(r.t)))$, where $W_S(r.t)$ is the sliding window of S at the arrival time of r.

3.2 Space Partitioning

Given a cluster with n processing units and a sliding window W, we define window partitioning as follows:

Space partitioning. The window partitioning P is denoted as $\mathcal{P} : \mathcal{W} \rightarrow \{\mathcal{W}_i, 1 \leq i \leq n\}$; it groups the elements of $\mathcal{W}$ into n window partitions $\mathcal{W}_i$ such that

$$(1)\ \forall i, j; i \neq j \Rightarrow W_i \cap W_j = \emptyset \qquad (2)\ \bigcup_{i=1}^{n} W_i = W.$$

In this work, we query a space partitioning in two different ways: a *point query* and a *range query*. In the former, we find the partition that contains a given point, and in the latter, we find all partitions that overlap in a given range of values, which we refer to as the *bounding partitions*.

Bounding partitions. Given the space partitioning $P : D \rightarrow \{D_i, 1 \leq i \leq n\}$ and the subspace $G (G \in D)$, the bounding partitions of G in P (P(G)) are the set of all partitions in P that overlap with G.

$$P(G) = \{D_i | D_i \cap G \neq \emptyset\}$$

4 ADAPTIVE STREAM *K*NN

Adaptive distributed stream kNN (ADS-kNN) is a distributed stream processing solution designed for processing kNN queries over data streams. ADS-kNN performs query execution and data partitioning in two separate and concurrent processes. Utilizing this design, these processes are synchronized only at critical points, which enables the system to achieve better system availability and response time.

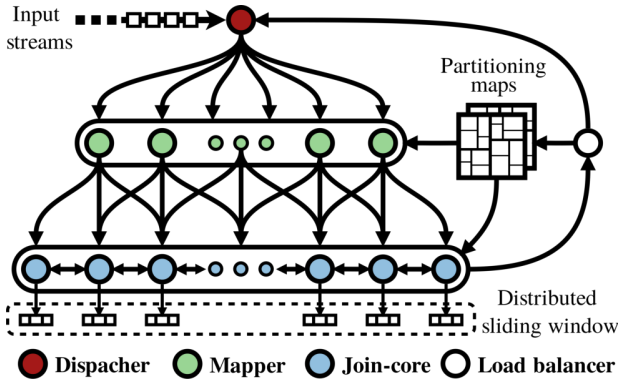


Figure 1: Overall architecture of ADS-kNN.

In this section, first, we introduce the ADS-kNN components and their responsibilities. Second, we describe the different stages of *k*NN query execution using ADS-kNN. Last, we present our adaptive space partitioning based on the proposed architecture.

4.1 ADS-kNN Structure

ADS-kNN consists of four different types of operators: a *dispatcher*, *mapper*, *join-core* and *load-balancer*. The dispatcher receives input tuples from both streams R and S and distributes them among the mappers. The mapper operators forward the tuples toward their corresponding join-cores according to a partitioning map, and the join-cores store the tuples in their local memory and generate the *k*NN results. Concurrently with tuple processing, the load-balancer continuously monitors the load distribution among the join-cores and proposes a new partitioning map if required. Figure 1 depicts the overall architecture of ADS-kNN.

Given n join-cores ($C_i, 1 \leq i \leq n$), we require the partitioning map (P), which decomposes the domain space into n subspaces ($P : D \rightarrow \{D_i, 1 \leq i \leq n\}$). For the given time t, the sliding window $W(t)$ is divided into n partitions $\{W_i(t) : 1 \leq i \leq n, \forall t_j \in W_i(t) \rightarrow t_j.p \in D_i\}$, and each partition $W_i(t)$ is stored in the join-core C_i.

4.2 *k*NN Query Process

Tuple processing in ADS-kNN consists of six stages: *distribution*, *mapping*, *storing*, *local search & neighbor querying*, *neighbor response*, and *aggregation*. The first two stages, distribution and mapping, are routing stages, in which tuples are forwarded to their corresponding join-cores. The remaining four stages are execution stages, in which the *k*NN results are generated. Depending on the partitioning map and tuple value, a join-core might need to query other join-cores to find the k nearest neighbors. Otherwise, the last two execution stages are omitted, and *k*NN query execution terminates after the *local search & store* stage. In our current design, there is no guarantee regarding the order of the output tuples. To generate a result output in the same order as the input arrival order, we must add an extra operator to buffer output tuples and propagate them in the correct order. Figure 2 illustrates the different stages of *k*NN processing.

Distribution. Tuple processing starts at the dispatcher, which receives all tuples and distributes them among the mappers. To reduce the communication overhead, the dispatcher groups the tuples into sets, which we refer to as *tasks*, and distributes the tasks

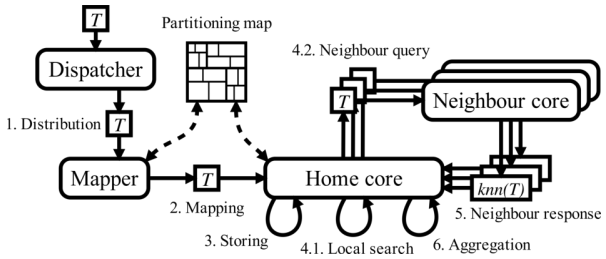


Figure 2: Stages of processing a tuple in ADS-kNN.

among the mapper operators instead of one single tuple at a time. The dispatcher arranges the tuples into tasks according to their arrival order, independently of which stream they belong to. As a result, each task may contain a mixture of tuples from streams R and S. To provide a uniform workload among the mappers, the dispatcher distributes the tasks in a round-robin order.

Mapping. For each tuple in a task message, the mapper operators forward it to its home-core, which is the join-core where the tuple must be stored. The mappers identify the home-cores by querying the partitioning map. Given task T and n join-cores, a mapper operator divides T into n subtasks ($T'_i, 1 \leq i \leq n$). The subtask T'_i includes all tuples of task T for which C_i is their home-core. At the end, the mapper forwards all subtasks to their assigned join-cores, even when a subtask is empty. Each subtask message contains information about the state of the sliding window at the moment when the task is created, such as the latest timestamps of each stream. We refer to this information as the *window status*. This information is needed by the join-cores to correctly maintain their local indexes. Therefore, we send all join-cores a subtask for each task created by the dispatcher, even when the subtask is empty.

Storing. At this stage, the join-cores update their local index and start performing *k*NN query execution. Each join-core maintains two individual indexes for its local partitions of streams R and S. Although indexing the content of stream R is not required for *k*NN query execution, the distribution of the tuples from stream R is needed to propose a balanced space partitioning. Thus, each join-core stores the tuples of stream R for the same criteria, time-based or count-based, as for the sliding window of stream S. The processing of tuples from stream S ends at this point.

Local search & neighbor query. For tuples from stream R, the join-cores obtain their local *k*NN results by querying their local index. Next, the join-cores determine whether their local results are sufficient to create the join result by querying the partitioning map. If there are no other partitions within the distance of the farthest local *k*NN from the local join-core partition in the partitioning map, the join-core knows that its local *k*NN is sufficient, and there is no need to query other join-cores. In this case, a join-core creates a result tuple using its local *k*NN results, and *k*NN query processing ends at this point. Otherwise, the join-core creates a neighbor request for join-cores that may have tuples closer than the local results. For each subtask and neighbor core, we create a single neighbor query message. Unlike the mapping stage, in which we may create an empty subtask message, neighbor request messages are generated only for neighbors whose responses are needed.

Neighbor response. For each tuple in a request message, a join-core queries its local index of stream S, collects all query results,

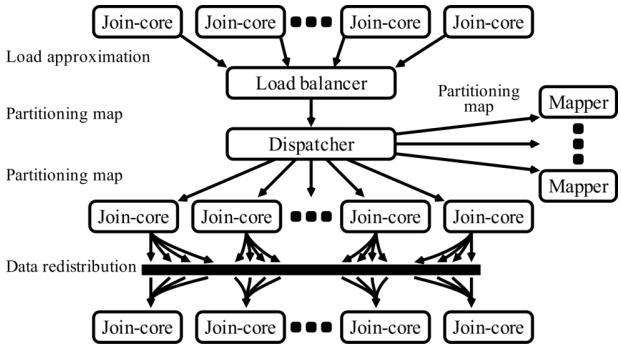


Figure 3: Stages of data repartitioning in ADS-kNN.

packs them into a single response message, and sends it to the requesting join-core.

Aggregation. In the last stage, the join-core that sent the neighbor queries collects all neighbor responses and generates the global *k*NN results.

4.3 Adaptive Space Partitioning

To maintain a balanced workload among the join-cores, it is necessary to update the partitioning map according to the tuples' distribution changes. We propose an adaptive space partitioning mechanism that consists of two operations, *workload monitoring* and *partitioning update*. The load-balancer continuously monitors the distribution of the workload among the join-cores, and if the distribution is unbalanced, the load-balancer initiates the partitioning update process.

Workload monitoring. The load-balancer constantly monitors the distribution of the workload among the join-cores to evaluate the efficiency of the current partitioning map. The workload monitoring is performed on an interval-by-interval basis, and the length of the intervals is defined as the number of task messages received by the join-cores. At the end of each interval, all join-cores submit their loads during the interval to the partition manager. For each interval, the load-balancer calculates the variance of the workloads among the join-cores. If the load distribution variance surpasses a defined threshold, the partition manager triggers the partitioning update process.

Partitioning update. ADS-kNN performs the partitioning update in two phases, *map generation* and *map replacement*. The load-balancer initiates the map generation phase by sending a load approximation request to the dispatcher. The dispatcher forwards this message to the next mapper in a round-robin order, and the mapper broadcasts this request to all join-cores. Upon receiving the load request, the join-cores create an approximation of their local workload and forward it to the load-balancer. The load-balancer collects all workload approximations and creates a new partitioning map accordingly, and then the first phase is complete.

The load-balancer starts the map replacement phase by sending the new partitioning map to the dispatcher. Before distributing the new partitioning map, the dispatcher submits a sync request to all mappers and join-cores. Synchronization is needed to ensure that all ongoing tasks, which are distributed based on the previous partitioning map, are completed before replacing the map with a new map and that we generate consistent results. The mappers and join-cores complete all unfinished tasks as they receive sync requests and send acknowledgments to the dispatcher. Next, the dispatcher broadcasts the new partitioning map to all mappers and join-cores. The mapper operations simply discard the old partitioning map, and any upcoming tuples are mapped based on the new partitioning map. The join-cores can perform data repartitioning in two modes: *instant* and *lazy*. In the former, the join-cores instantly repartition the tuples after receiving the new partitioning map. The join-cores query the new partitioning map to find the new home-cores of their locally stored tuples and send the tuples to their new home-cores. At the end, they discard the old partitioning map and resume task processing as normal. In the latter mode, the join-cores do not redistribute the tuples, and instead, resume the join operation until the old partitioning map expires. In this scenario, the join-cores maintain multiple versions of the partitioning map. All newly arriving tuples are partitioned according to the latest version. However, whenever a join-core checks for overlapping partitions with its local *k*NN results, it checks all available partitioning maps and sends requests to all overlapping join-cores in all versions. A partitioning map expires when the last tuple partitioned using this partitioning map is eliminated from the system. At this point, the partitioning map can be eliminated by the join-cores. Figure 3 illustrates the different stages of data repartitioning in ADS-kNN.

5 ADS-KNN IMPLEMENTATION

In this section, we present the implementation details of our approach, including our stream processing framework, indexing approaches, and unified mapper-joiner operator, which we propose for better resource utilization. Moreover, we discuss task synchronization and result correctness in ADS-kNN.

5.1 MPI-based Stream Processing

We design a lightweight stream processing framework that provides the flexibility and performance that we require in implementing ADS-kNN. Using our own stream processing system allows us to study the effect of different design decisions, which provides us with new insights about challenges and limitations to overcome. Despite the availability of several stream processing frameworks, such as Flink, Storm and Spark Streaming, we opted to design our own stream processing system. We conducted a detailed study of existing stream processing frameworks to implement our approaches in the early stages of our research, but we perceived substantial performance limitations as also observed by other researchers [20, 42]. The reason is that none of the available frameworks are optimized for the type of workload that we require in this work. The mentioned frameworks are based on the massive parallelism of independent tasks, and their target platforms are multinode clusters. Consequently, these platforms result in suboptimal performance for ADS-kNN, which relies on frequent low-latency queries and operator coordination.

To maximize performance, each operator in our stream processing system is a process with a dedicated central processing unit (CPU) core. Therefore, the maximum number of operators is bounded by the number of available CPU cores. Operators use MPI to communicate with each other. All messages in our system are transmitted using nonblocking communication, which enables better resource utilization by overlapping communication with

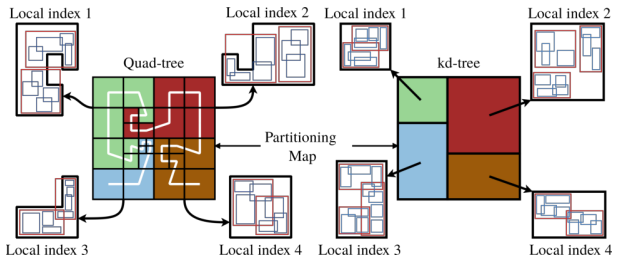


Figure 4: Space partitioning: quadtree and *k*d-tree.

computation. The message types are encoded as MPI message tags to ensure that messages of the same type are always received in the same order as they are transmitted, which is crucial for our system to generate the correct results.

5.2 Indexing and Workload Approximation

To accelerate *k*NN queries, the join-cores index their local window partitions with an R*-tree. We opt to use an R*-tree because it is capable of incremental updates and adapts to changes in the data distribution. In addition to accelerating queries, the join-cores use their local indexes to generate workload approximations, which is required for creating a partitioning map.

An R*-tree is a tree-based data structure designed for indexing multidimensional data, which stores the data elements in leaf nodes. It groups nearby objects at each level into a minimum bounding box and uses these boxes as objects for the next higher level. The minimum bounding boxes at each depth of the R*-tree represent an approximation of the indexed tuple distribution. Therefore, join-cores do not require extra computation to calculate the distribution approximation. The greater is the depth, the finer are the bounding boxes, and therefore, the more accurate is the approximation. The decision of which precision level of the distribution approximation should be used for creating the domain partitioning trades off the partitioning efficiency and processing overhead. The finer the distribution approximation that we use is, the more accurate our partitioning is, but the more costly it is to transmit the data and create a partitioning map.

The load distribution in ADS-kNN is dependent on the distributions of both data streams. The tuples from stream R are the *k*NN queries, and the tuples from S are the data points in the domain. To generate effective domain partitioning, we must consider the distribution of both the queries and the data points. Therefore, we maintain two sliding windows in ADS-kNN, one for each stream, even though the sliding window of stream S is not needed in the *k*NN join process. We opt to accept the extra cost of maintaining a sliding window to obtain the benefit of more efficient domain partitioning. For this reason, the join-cores index their local sub-windows for both the R stream and the S stream into two separate R*-trees.

5.3 Space Partitioning Data Structure

We use three space partitioning data structures in this work: a *k*d-tree, quadtree and grid file. Although both the *k*d-tree and the quadtree are data structures for indexing multidimensional data points, there are important differences between these two data structures. A *k*d-tree partitions data points, but a quadtree partitions the space. Therefore, the boundary of partitions is set according to data points using a *k*d-tree, but the quadtree sets boundaries according to the space. Moreover, a *k*d-tree is a balanced tree, and we can control the number of leaf nodes, while a quadtree is not balanced, and we cannot control the number of leaf nodes. For this reason, to use a quadtree as a space partitioning method, we must create more leaf nodes than join-cores and use a space-filling curve. In contrast to adaptive space partitioning using tree-based data structures, the grid file divides space into equally sized cells using a flat data structure. Thus, we opt to implement all these data structures as partitioning maps for ADS-kNN, and we study the performance of each approach in the application of a distributed stream join. Figure 4 illustrates the tree-based partitioning schemes and the local indexing approach used in ADS-kNN.

***k*d-tree:** Given the load approximation and N join-cores, our objective is to create a *k*d-tree with N leaf nodes with a uniform workload among them. We start at the root node with the entire space D and N join-cores. At each inner node, we calculate the standard deviation (σ) of the load distributions along each axis and divide the space perpendicular to the axis with the highest σ, such that the loads at each child node are $\lceil N/2 \rceil$ and $\lfloor N/2 \rfloor$. We continue the process for each child node until we reach the leaf nodes. Each leaf node is assigned to a single join-core.

quadtree: To create a partitioning map using a quadtree, we start at the root node, which includes the entire domain and workloads. We expand the inner nodes of the quadtree until the load at each leaf node is less than a predefined threshold, which is the granularity of partitioning. The finer this threshold is, the more uniform the load balancing is and the more expensive the partitioning query is. Therefore, choosing the right partitioning granularity is important for the performance of our system. We linearize the divided space using a Hilbert space-filling curve, and then we split the Hilbert curve into N parts such that the load assigned to each part is the same. At the end, each portion of the Hilbert curve is assigned to a single join-core.

Grid file: Creating space partitioning using the grid file is easier than that using *k*d-tree and quadtree. We divide the domain space into a uniform grid and calculate the load on each grid cell. To distribute grid cells among join-cores, we linearize the grid using the Hilbert space-filling curve and then divide the Hilbert curve into N sections, similar to quadtree.

5.4 Unified Mapper/Joiner Architecture

A simple approach to implementing ADS-kNN in our stream processing framework is to fix the number of mappers and join-cores in the system and statically assign a dedicated process to each of them. However, there are two drawbacks to this approach. First, there is no fixed optimal value for the number of mappers and join-cores, and we must empirically search for the optimal configuration for each individual streaming application and system setup. Second, a static system configuration cannot properly work with dynamic data distribution changes. For example, if the input data distribution changes in a way that results in more neighbor queries, we will require more computing power for the join-cores to resolve all queries.

To address these concerns, we propose a new operator, the *mapper-joiner*, which functions as both a mapper and a join-core.

Using n mapper-joiners is logically the same as having n mappers and n joiners. Using the mapper-joiner operator, we rely on adapting the execution time that we spend on each operation rather than adapting the number of processes. In these settings, when the neighbor querying rate increases, the mapper-joiners automatically dedicate more time for the join operation, and likewise, when the data partitioning is more costly, the mapper-joiners spend more time on mapping operations. Therefore, we can allocate all available computing resources to the mapper-joiners independently of the application and system configuration, and the mapper-joiner operators dynamically utilize computing resources as the data distribution changes.

5.5 Task Synchronization

We must ensure that two conditions hold true for a join-core to provide a correct response to kNN requests. First, a join-core must have received all subtasks prior to the task to which a given kNN request belongs. Second, a join-core should not eliminate a tuple unless it does not match any future kNN requests from any other join-core. To ensure that these conditions hold true in ADS-kNN, we propose two mechanisms: *buffered kNN requests* and *relaxed window synchronization*.

Task execution is not a synchronized process in ADS-kNN, and join-cores may process tuples that belong to different tasks at a point in time. Therefore, a join-core might receive a kNN request from a task that is not received by the join-core. In this scenario, the join-core cannot provide a correct response unless it receives its corresponding subtask. To address this issue, we propose a buffered kNN request mechanism. Thus, each join-core maintains a buffer for kNN requests that cannot be resolved instantly. When a join-core receives a new kNN request, it verifies whether it can instantly provide a correct response; if it can it processes the request as normal, and no further actions are required. Otherwise, the join operator stores the kNN request in its request buffer and resumes the join operation. To resolve buffered requests, the join-cores probe their local buffers upon receiving every new subtask and verify if any request can be resolved.

Over time, tuples expire in the sliding window, which means that these tuples will not match any other tuple due to the window constraints. It is important to eliminate expired tuples to maintain the indexing data structures in an efficient state. However, the early elimination of tuples may result in incorrect kNN results. Therefore, a join-core must be aware of other join-cores' task execution states to correctly eliminate tuples. To address this issue, we propose an implicit task synchronization mechanism that does not require any extra messaging among join-cores. Implicit task synchronization is based on limiting the number of active tasks in the system. A task is considered to be active when the corresponding mapper operator divides it into subtasks until all subtasks are processed and the join results are generated. By limiting the number of active tasks, a join-core can implicitly deduce the correct sliding window boundaries, such that it can eliminate tuples without causing incorrect results.

We limit the number of active tasks to $2n$, where n is the number of join-cores. As mentioned in Section 4, each subtask includes the window status information. The join-cores maintain a list of window statuses with a length of $2n$. When a new subtask arrives, a join-core inserts the new window status into the list and removes the least recent window status. Next, the join-core deletes all expired tuples based on the assumption that the task corresponding to the removed window status is finished, and it can eliminate tuples from its local index. To ensure that there are no more than $2n$ active tasks in the system, we propose a solution based on the unified mapper-joiner operator. Consider a system consisting of n mapper-joiner operators; we therefore have n mappers ($M_i, 1 \leq i \leq n$) and n join-cores ($J_i, 1 \leq i \leq n$). Additionally, each task T_j (the $j-th$ task in the system) is divided into n subtasks $T'_{j,k}$ ($1 \leq k \leq n$). The mapper M_i does not distribute the subtasks of task T_j unless the join-core J_i has finished processing all its subtasks prior to task T_{j-n}. If this condition holds true, and considering that the join-cores receive subtasks in order, a join-core can deduce that when it receives the subtask $T'_{j,k}$, all join-cores have finished processing the subtasks of task T_{j-2n}. Therefore, there will be no requests from tasks prior to T_{j-2n}, and the join-core can eliminate tuples without causing incorrect kNN results.

6 COST ANALYSIS

Here, we provide an analytic comparison between the ADS-kNN join and distributed stream kNN join based on round-robin partitioning with respect to computational complexity, communication cost and memory consumption.

6.1 Round-robin Partitioning

As a baseline of our approach, we implement a distributed $R \ltimes_{kNN} S$ based on round-robin partitioning, which we refer to as RR-kNN. In this approach, tuples are distributed among the join-cores in a round-robin order rather than using a space partitioning-based approach. To find the kNN of a tuple, each join-core in the system finds its local kNN results and sends them to the tuple's home-core. The home-core gathers all local kNN results and generates the global kNN. To accelerate the process of finding a local kNN, each join-core indexes its local window partition in the R*-tree, similar to ADS-kNN.

6.2 Computational Complexity

To compare the computational complexity of ADS-kNN and RR-kNN, we estimate the average cost of processing a single tuple from each stream R and S using either solution.

ADS-kNN: Tuples from streams R and S are processed differently in ADS-kNN. A tuple from stream S is processed in the following three steps: (1) query the partitioning map to find its home-core, (2) insert the tuple into the home-core's local index, and (3) delete the tuple from the index when it is expired. For a uniform tuple distribution among the join-cores, Equation 1 represents the cost of processing a single tuple from stream S, where δ_i and δ_d are the cost of the insertion operation and cost of the deletion operation, respectively, in each local index, and C_p is the cost of querying the partitioning map.

$$C_s = C_p + \delta_i + \delta_d \tag{1}$$

Processing a tuple from stream R requires four more steps to find its kNN: (1) find the local kNN in its home-core, (2) query the partitioning map to find the overlapping join-cores, (3) find the kNN in the neighboring join-cores, and (4) merge the local kNN results and find the global kNN. Let α be the average number of

neighbor queries per tuple, and let C_m be the cost of merging the local kNN results. Equation 2 represents the cost of processing a single tuple from stream R, where δ_s is the cost of the kNN query in each local index.

$$C_r = 2 \cdot C_p + \delta_i + \delta_d + (1 + \alpha) \cdot \delta_s + C_m \quad (2)$$

RR-kNN: To process the tuples from stream S using RR-kNN, we require two operations: (1) insert the tuple into the local index of its home-core, and (2) delete it from the index when it expires. The tuples from stream R are processed as follows: (1) find the local kNN in each join-core, and (2) merge all local results to find the global kNN. Equations 3 and 4 represent the cost of a single tuple of stream S and stream R, respectively, using RR-kNN, where C'_m is the cost of merging the local kNN results in RR-kNN.

$$C'_s = \delta_i + \delta_d \quad (3)$$

$$C'_r = n \cdot \delta_s + C'_m \quad (4)$$

The cost of processing a tuple from stream S is almost the same in both solutions, considering that the cost of querying the partitioning map is logarithmic in the number of join-cores. Thus, this cost is negligible in comparison with the cost of the update operations in the R*-tree. The main drawback of using round-robin partitioning is that the complexity of kNN queries increases linearly with an increasing number of join-cores. Therefore, utilizing more join-cores is not effective in improving the system performance. ADS-kNN limits the neighbor queries by utilizing a space partitioning. Thus, the cost of kNN computation does not increase by increasing the number of join-cores.

6.3 Communication

To compare the communication overheads of ADS-kNN and RR-kNN, we measure the average amount of data transfer among the operators needed to process a single tuple using either approach. We estimate the data transfer size in terms of the average tuple size (γ_t). The communication cost of processing a tuple has two parts: (1) forwarding the tuple to its home-core and (2) processing kNN queries. We measure the communication cost of each approach as follows:

ADS-kNN: We must transmit a tuple twice to take it to its home-core, from the dispatcher to the mapper and from the mapper to the join-core. To find the kNN of a tuple, a join-core sends queries to α join-cores, and they return k tuples in response. Therefore, the communication cost of finding the kNN is $\alpha \cdot (1+k) \cdot \gamma_t$. The average communication cost of each tuple from stream R and stream S is represented in Equations 5 and 6, respectively.

$$V_r = 2 \cdot \gamma_t \quad (5)$$

$$V_s = 2 \cdot \gamma_t + \alpha \cdot (1 + k) \cdot \gamma_t \quad (6)$$

RR-kNN: Each tuple is directly transferred to join-cores in round-robin order, so we must transmit a tuple only once to reach its home-core. To find the kNN of a tuple, we send n queries to all join-cores, and each join-core returns k tuples in response. The total communication cost for processing the kNN is $n \cdot (1 + k) \cdot \gamma_t$. The average communication cost of processing a tuple from each stream R and stream S is represented in Equations 7 and 8, respectively.

$$V'_r = \gamma_t \quad (7)$$

$$V'_s = \gamma_t + n \cdot (1 + k) \cdot \gamma_t \quad (8)$$

Comparing the communication costs of ADS-kNN and RR-kNN, we observe that forwarding tuples toward their home-cores in ADS-kNN is twice as costly as in RR-kNN, but the difference is not significant. However, the cost of finding the kNN using RR-kNN is remarkably higher than that of ADS-kNN. In the case of ADS-kNN, the local kNN result in the home-core is equivalent to the global kNN result if we use an effective partitioning map. In these scenarios, there is no need to transfer any data among operators. However, the communication cost of finding a tuple's kNN using RR-kNN increases linearly with an increasing number of join-cores, similar to the computational complexity. Therefore, RR-kNN requires transferring n/α times more data than ADS-kNN.

6.4 Memory Footprint

To compare the memory footprints of ADS-kNN and RR-kNN, we consider only the memory space required for tuple indexing. Although there are other memory buffers required in both approaches, the memory overheads of these buffers are negligible in comparison with the size of the R*-trees that we use for indexing tuples. Likewise, the partitioning map is a comparably smaller data structure than the R*-tree, considering that the memory footprint of the partitioning map is linear in the number of join-cores, while the R*-tree's size is on the order of the number of tuples.

The relative amount of memory space needed for indexing tuples in RR-kNN and ADS-kNN depends on the input ratio between stream R and stream S. ADS-kNN requires indexing the sliding windows of both streams R and S, while RR-kNN indexes only the tuples of stream S. ADS-kNN indexes the tuples of stream R for the same period as stream S, which is needed for balancing the workloads. In the case of an equal input rate for both streams, ADS-kNN requires simply twice the memory space of RR-kNN—two equally sized R*-trees in ADS-kNN, in comparison with a single R*-tree in RR-kNN. Let ρ_r and ρ_s be the input rates of stream R and stream S, respectively. ADS-kNN requires $1 + \frac{\rho_r}{\rho_s}$ times more space for indexing tuples than RR-kNN. Because the space complexity of R*-tree is linear with regard to the number of elements, the index of stream R is $\frac{\rho_r}{\rho_s}$ times larger than that of stream S. Therefore, the total indexing memory footprint of ADS-kNN is $1 + \frac{\rho_r}{\rho_s}$ times the index size of stream S.

7 EVALUATION

In this section, we present a set of experiments to benchmark the efficiency of the approaches presented in this paper. For our evaluations, we use a four-socket NUMA workstation equipped with 14-core Intel Xeon E7-4850v3 processors and 128 GB of DDR3 memory on each socket. We ran each experiment three times and reported the average over three runs.

To maximize performance, we deploy up to 56 operators in our stream processing framework using our hardware (one operator per CPU core). In every evaluation, we dedicate two CPU cores for the emitter and measurement operators. The former buffers the input tuples into the main memory and sends them to the dispatcher operator, and the latter gathers statistics about the system performance. Therefore, 54 operators are available for performing kNN queries.

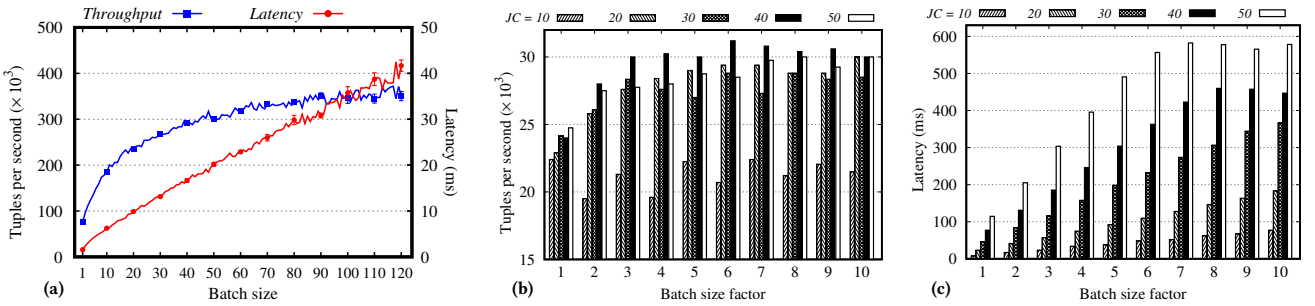


Figure 5: a) Throughput and latency of ADS-kNN for various batch sizes. b) Throughput of RR-kNN for various batch size factors and join-cores. c) Latency of RR-kNN for various batch size factors and join-cores.

We study two different forms of kNN join, namely, two-way join and self-join. In the former, R and S are two distinct streams, and in the latter, R and S are identical streams. Our experiments are conducted on both real-world datasets and synthetic datasets. Our real-world datasets are *TWEETS* and *NY-TAXI* [35]. *TWEETS* is a set of geospatially tagged tweets from North America gathered from January 2018 to March 2018; it is utilized to benchmark the self-join operator. *NY-TAXI* is a set of taxi rides in New York City in 2017. We use the rides' pick-up location and drop-off location as the tuples of stream R and stream S, respectively, to perform a two-way stream join. To provide more controlled studies of our data repartitioning methods, we generate a synthetic dataset, in which the tuple distribution shifts with a controlled parameter. Using our synthetic dataset, we can study how our data repartitioning methods perform during data distribution changes.

7.1 Batch Size

In the first experiment, we empirically determine the optimal batch size for both ADS-kNN and RR-kNN. We evaluate each approach for various numbers of batch sizes and measure their performance in terms of both throughput and latency. For this experiment, we employ the two-way join operator using the *NY-TAXI* dataset and set the window size and k to 5×10^6 and 25, respectively.

Figure 5(a) illustrates the throughput and latency of ADS-kNN using various batch sizes ranging from 1 to 128 and 52 mapper-joiner operators. As we increase the batch size, we observe that the ADS-kNN throughput increases rapidly at the beginning. The gains gradually decrease, and eventually, the increasing batch size does not improve the system throughput, and the performance curve becomes flat. The results show that the system latency follows a different pattern than throughput. The system latency increases almost linearly with an increase in batch size. Selecting an optimal batch size is a trade-off between throughput and latency. Depending on the application, we may prefer better result latency at the cost of lower throughput, or vice versa. In the remainder of this work, we use a batch size of 100 for all our experiments using ADS-kNN. Nevertheless, it is viable to use smaller batch sizes in latency-critical applications. For instance, using a batch size of 20 instead of 100 reduces the system latency to almost one-fourth, while it reduces the system throughput by only 20%.

In the case of RR-kNN, we set the batch sizes as an integer multiple of the join-cores, which we refer to as the *batch size factor*. We use this configuration to provide a uniform workload among the join-cores using RR-kNN. Figures 5(b) and (c) illustrate the throughput and latency of RR-kNN using various batch size factors and mapper-joiner operators. The results show that the batch size does not make a significant change in RR-kNN throughput. As we increase the batch size factor from 1 to 3, the system throughput increases by 20% on average. However, the system performance does not improve any further by using a batch size factor higher than 3. In terms of the system latency, increasing the batch size factor has a major drawback. The system latency increases almost linearly with an increase in batch size. Because the performance gain of using a batch size factor of 3 is not significant in comparison with a batch size factor of 2, we opt to use a batch size factor of 2 for the remaining experiments to attain lower latency.

7.2 Space Partitioning

We now study the efficiency of our space partitioning based on data distribution approximation using three data structures that we implement in this work, kd-tree, quadtree and grid file. To evaluate the efficiency of our approach, we implement a space partitioning method based on data preprocessing, which we refer to as the baseline. The baseline method preprocesses the input data and creates a space partition using quadtree. We conduct two experiments: first, we study how each space partitioning scales with different numbers of join-cores; second, we study the workload distribution among the join-cores using each space partitioning method. For this experiment, we use the self-join operator and the *TWEETS* dataset and set the window size and k to 5×10^6 and 100, respectively.

Figure 6(a) shows the throughput of ADS-kNN for various numbers of join-cores ranging from 2 to 52 using four different space partitioning methods, including the baseline method and three approximate-based methods, grid file, quadtree and kd-tree. In general, all approaches yield similar performance for join-cores between 2 and 22, while the performance becomes more divergent for join-cores exceeding 24. This finding indicates that the partitioning map becomes more influential when we utilize more join-cores in our system. In this experiment, quadtree outperforms the other two data structures, kd-tree and the grid file, by 5% on average. Overall, our real-time space partitioning approach demonstrates an adequate performance, which results in only 5% less throughout than the preprocessing-based solution on average.

In the second experiment, we perform distributed stream kNN join based on ADS-kNN and measure the load distribution among join-cores. We measure the load of a join-core as the total number of insert and search queries that each join-core performs over a fixed

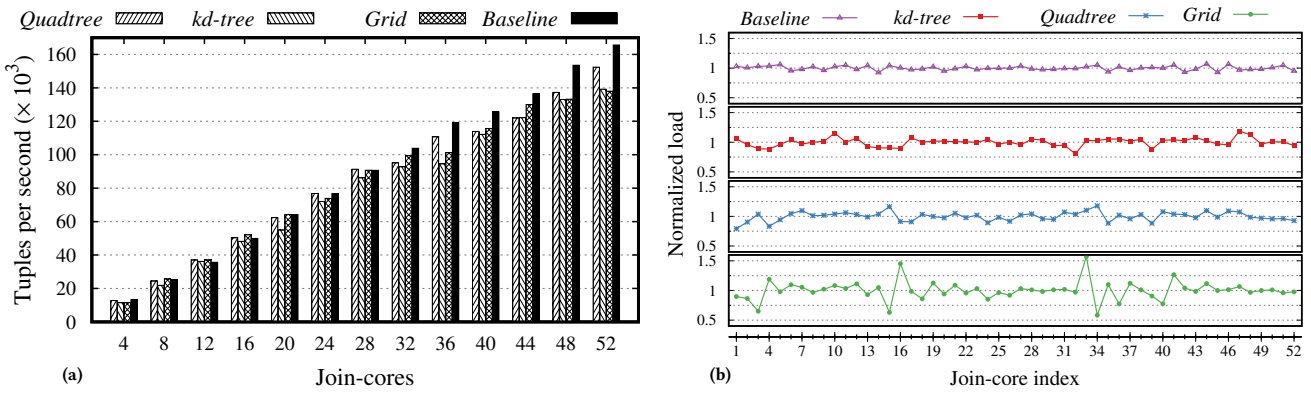
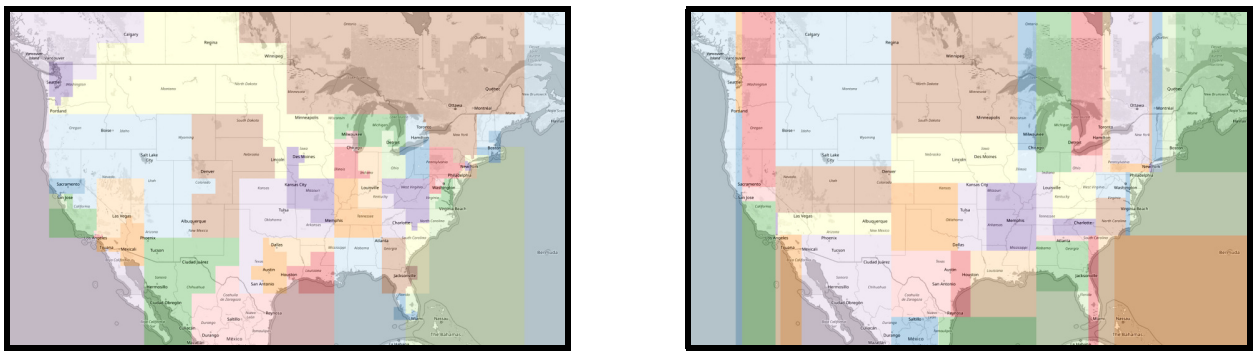


Figure 6: a) ADS-kNN throughput using *k*d-tree, quadtree and grid file for various numbers of join-cores. b) Normalized workload distribution over join-cores in ADS-kNN using *k*d-tree, quadtree and grid file, where load equal to 1 is the average load.



(a) quadtree **(b) kd-tree**

Figure 7: Partitioning the *TWEETS* dataset.

period of time. Figure 6(b) presents the normalized load distribution among 52 join-cores using quadtree, *k*d-tree, grid file and our baseline. The baseline results in the most uniform load distribution. The workload distribution among join-cores using the two tree-based partitioning maps, *k*d-tree and quadtree, is reasonably uniform, and the majority of workloads are between 80% and 120% of the average. The worst-performing data structure in terms of load distribution is the grid file, which cannot properly handle densely populated areas. For the remainder of the evaluations, we use quadtree as a space partitioning map because of its higher average throughput. Figures 7(a) and (b) illustrate the quadtree partitioning map and the *k*d-tree partitioning maps, respectively, for the same experiment. These results provide better insight into the workload distribution and how each approach partitions the space.

7.3 Scalability

The objective of this experiment is to study the scalability of the approaches presented in this paper with respect to the number of join-cores. For this reason, we evaluate the stream *k*NN join using both ADS-kNN and RR-kNN for various numbers of join-cores and measure the throughput and latency. In this experiment, we employ the two-way join operator using the *NY-TAXI* dataset and set the window size and k to 5×10^6 and 25, respectively.

Figure 8 (a) compares the throughputs of ADS-kNN and RR-kNN using different numbers of join-cores, ranging from 1 to 52. Using a single join-core, RR-kNN outperforms ADS-kNN by 30% because of the mapping overhead of the tuple distribution in ADS-kNN. However, as we increase the number of join-cores, the RR-kNN throughput scarcely improves, and RR-kNN using 52 join-cores is only 40% faster than the single join-core configuration. In contrast, the distributed *k*NN stream join using ADS-kNN scales reasonably well as we increase the number of join-cores; using 52 join-cores, and ADS-kNN yields a throughput that is more than 30 times higher.

Figure 8 (b) illustrates the system latency for the same experiment. The results indicate that ADS-kNN scales well in terms of latency. The ADS-kNN latency does not change significantly for different configurations and remains level. In contrast, the latency results for RR-kNN increase greatly with an increasing number of join-cores, such that using the highest evaluated number of join-cores, 52, produces almost 240 times greater latency than using a single join-core. The RR-kNN latency increases with an increasing number of join-cores for two reasons. First, each join-core requires the collection of local *k*NN results from additional join-cores, and second, the cost of merging the local *k*NN results increases with an increasing number of join-cores.

These experiments show that ADS-kNN is a scalable approach in terms of both throughput and latency. Using the highest evaluated number of join-cores, 52, ADS-kNN results in approximately 30 times higher throughput than the single join-core configuration, and the system latency decreases by approximately 5%. In contrast, the results indicate that round-robin partitioning is not a scalable data distribution approach for the application of distributed *k*NN

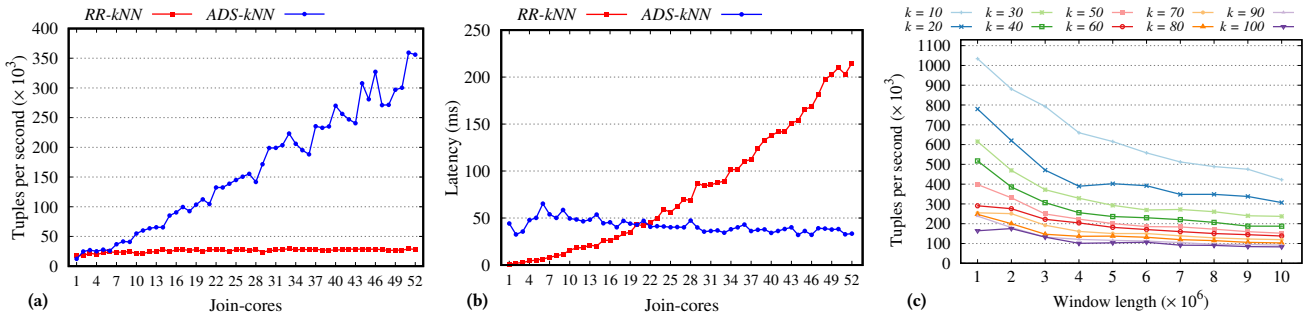


Figure 8: a) Throughput comparison between ADS-kNN and RR-kNN using different numbers of join-cores. b) Latency comparison between ADS-kNN and RR-kNN using different numbers of join-cores. c) ADS-kNN throughput for various window sizes and values of *k*.

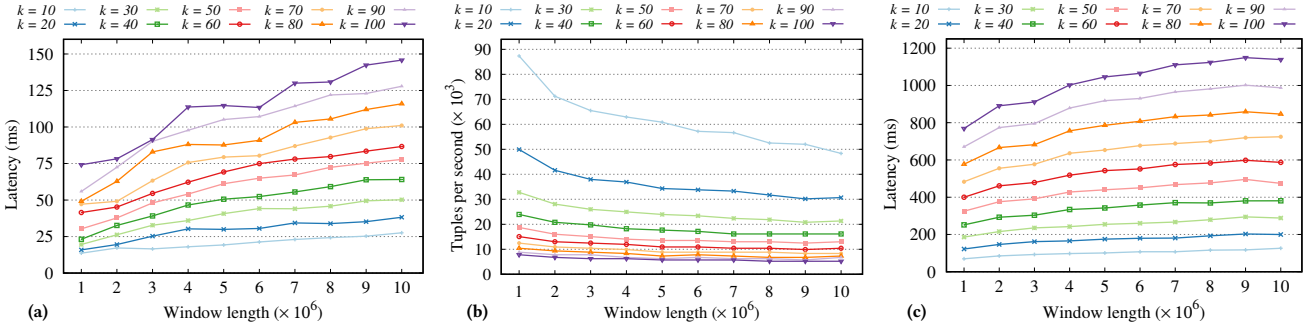


Figure 9: a) ADS-kNN latency for various window sizes and values of *k*. b) RR-kNN throughput for various window sizes and values of *k*. c) RR-kNN latency for various window sizes and values of *k*.

join. The additional computational complexity of *k*NN queries and the communication overhead resulting from using a larger number of join-cores restrict the scalability of round-robin partitioning in this application.

7.4 Window Size and *k*

In this experiment, we investigate the effect of two parameters, the window size and k, on the performance of ADS-kNN and RR-kNN. We evaluate the distributed *k*NN join for various values of k and window sizes using both ADS-kNN and RR-kNN, and we compare the system performance in terms of throughput and latency. In this experiment, we use the two-way join operator using the *NY-TAXI* dataset.

Figures 8 (c) and 9 (a) show the throughput of ADS-kNN and RR-kNN, respectively, for different values of k and window sizes. We observe a similar pattern in the changes in performance for both ADS-kNN and RR-kNN; however, the magnitude of the changes differs between these two approaches. As we increase the value of k, the performance of both approaches decreases sharply and then decreases gradually. ADS-kNN's throughput declines to 25% on average as we increase the value of k from 10 to 100. In the same comparison, RR-kNN's throughput decreases to one-fifth. Increasing the window size also decreases the performance of both approaches, although the performance losses are more linear with the size of the sliding window, and we do not observe sharp drops in the performance of either approach. On average, the throughput of ADS-kNN is 11 times higher than that of RR-kNN over the evaluated configurations.

Figures 9 (b) and (c) illustrate the result latency of ADS-kNN and RR-kNN, respectively, for the same experiment. The result latency follows a similar pattern as the throughput. Both ADS-kNN and RR-kNN result in higher latency as we increase the value of k or the window size. However, the value of k affects the performance of RR-kNN more strongly than that of ADS-kNN. When increasing the value of k from 10 to 100, the latency of ADS-kNN increases 5 times on average. In the same comparison, RR-kNN's throughput increases 7 times on average. The results indicate that the result latency of RR-kNN is 10 times higher than that of ADS-kNN on average over the evaluated configurations.

7.5 Dynamic Distribution

In this experiment, we study the space repartitioning methods presented in this paper, lazy and instant repartitioning. To gain better control over the data distribution changes, we generate a synthetic workload for this experiment, which is based on shifting the two-dimensional Gaussian distribution. We evaluate ADS-kNN using multiple shifting distributions with different shifting paces, and we analyze the efficiency of each repartitioning method. This experiment is based on a self-join operator; we set the window size and k to 5×10^6 and 25, respectively.

The shifting distribution consists of three phases. In the first phase, the tuples are generated according to the fixed Gaussian distribution $\mathcal{N}((0.5, 0.5), (0.125, 0.125))$ ($\mu = (0.5, 0.5)$, $\sigma^2 = (0.125, 0.125)$). During the middle phase, the distribution of the tuple values linearly shifts from $\mathcal{N}((0.5, 0.5), (0.125, 0.125))$ to $\mathcal{N}((0.5, r +$

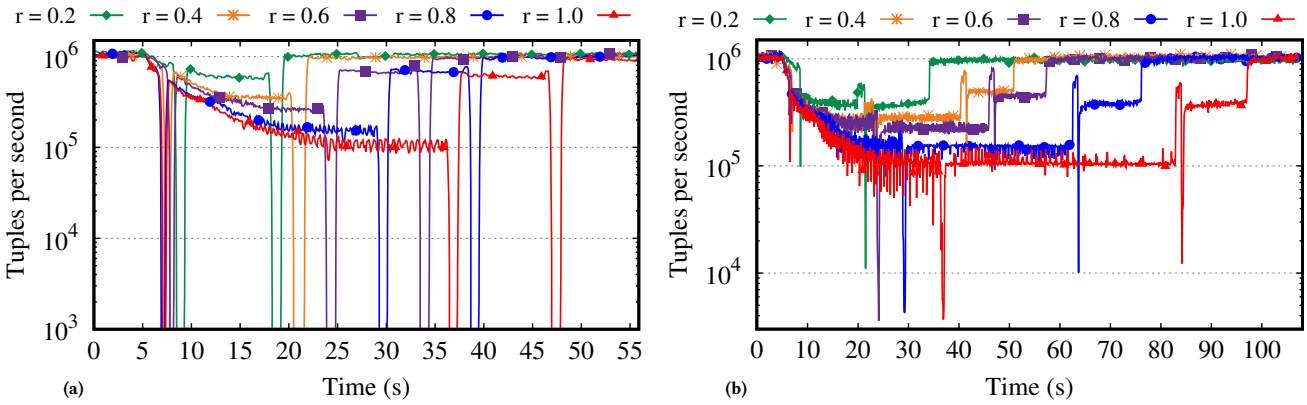


Figure 10: System performance during data repartitioning in ADS-kNN: (a) instant, (b) lazy.

0.5), (0.125, 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 $\mathcal{N}((0.5, r + 0.5), (0.125, 0.125))$. We evaluate ADS-kNN using 5 different shifting paces, from 0.2 to 1, and we measure the system throughput every 0.1 s.

Figure 10 (a) illustrates the throughput of ADS-kNN using instant repartitioning over the shifting distribution with various shifting paces. When the distribution shift begins (after approximately 5 s), the system throughput starts to decline. Eventually, the load balancer detects an imbalanced workload distribution and triggers the data repartitioning operation, and the join operation resumes. In general, the faster the data distribution changes, the lower the performance during the distribution changes and the later the system recovers. For the two slowest evaluated shifting paces, ADS-kNN performs two repartitioning operations, while other scenarios require three repartitioning operations. Each data repartitioning operation blocks the join operation for approximately 1 s, which is correlated with the sliding window size and the number of tuples to be redistributed. At the end, the system performance recovers to the normal level when the data distribution stops shifting.

Figure 10 (b) illustrates the results of the experiment using lazy repartitioning. In general, we observe that it takes more time to recover to a balanced workload using lazy data repartitioning than the instant variant. Instead of blocking the join operation to perform data redistribution, the join-cores continue the operation with multiple partitioning maps. Therefore, we observe only a few short performance drops during lazy repartitioning instead of join operation interruptions. However, the performance of lazy repartitioning is lower for a shifting data distribution compared with instant repartitioning. When an older partitioning map expires, the system performance increases. If the load balancer finds an unbalanced workload distribution, it triggers another data repartitioning operation. This process continues until the data distribution stops shifting and ADS-kNN finds a balanced workload.

8 CONCLUSIONS

In this paper, we presented ADS-kNN, a solution for high-performance distributed kNN join for data streams. We proposed a multi-stage query execution plan that enables scalable and low-latency kNN processing. This processing is achieved by enabling join-cores to perform frequent low-latency queries with each other and by overlapping the communication and computation stages. Furthermore, we proposed an adaptive data partitioning mechanism that dynamically distributes the workload among join-cores to maintain the system in an efficient state. To evaluate the efficiency of our approach, we conducted an extensive evaluation and compared our data partitioning approach to the mainstream method, round-robin partitioning. The results indicate that ADS-kNN is a scalable approach for the application of distributed stream kNN join. Utilizing 52 join-cores, ADS-kNN had a throughput more than 30 times higher than that using a single join-core. Moreover, ADS-kNN significantly outperformed the round-robin partitioning-based approach in terms of both throughput and latency. ADS-kNN achieved 12 times higher throughput than the round-robin-based approach using the highest evaluated number of join-cores.

The directions for our future work are twofold: (1) to design and develop a more advanced data partitioning approach and (2) to extend our stream processing framework. In this work, we presented a space partitioning solution that considers only the distribution of tuples among the join-cores in distributing tuples among the join-cores. This approach can be improved further by considering the queries among join-cores. Therefore, the partitioning map set partitions boundaries in a way that results in fewer communications. Moreover, our data repartitioning method can be further improved to reduce the number of tuples transferred during repartitioning by maximizing the overlap between the previous space partitioning and the new space partitioning. Another direction for our future work is to extend our stream processing framework to support a wider range of stream processing applications. The current communication patterns are designed according to the operators needed in this paper. Providing more extensive communication patterns among operators enables our stream processing framework to support other types of operations. Furthermore, we can extend our stream processing framework by providing better fault detection and recovery mechanisms to support multinode computing clusters.

9 ACKNOWLEDGMENTS

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

REFERENCES

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop GIS: A high performance spatial data warehousing system over Mapreduce. *VLDB*, pages 1009–1020, 2013.

[2] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: Adaptive query workload aware partitioning of big spatial data. *VLDB*, pages 2062–2073, 2015.

[3] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang. SparkGIS: Resource aware efficient in-memory spatial query processing. In *SIGSPATIAL*, pages 28:1–28:10, 2017.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, page 28–38, 2015.

[5] B. Chandramouli, J. Goldstein, M. Barnett, et al. Trill: A high-performance incremental query processor for diverse analytics. *VLDB*, pages 401–412, 2014.

[6] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W. Lee, and E. Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *IEEE Transactions on Knowledge and Data Engineering*, pages 925–938, 2016.

[7] Z. Cheng and J. Shen. Just-for-Me: an adaptive personalization system for location-aware social music recommendation. In *Proceedings of international conference on multimedia retrieval*, pages 185–192, 2014.

[8] Danzhou Liu, Ee-Peng Lim, and Wee-Keong Ng. Efficient k nearest neighbor queries on remote spatial databases using range estimation. In *Proceedings 14th International Conference on Scientific and Statistical Database Management*, pages 121–130, 2002.

[9] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *SIGMOD*, page 2471–2486, 2020.

[10] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *SIGMOD*, pages 1123–1134, 2008.

[12] L. Golab, S. Garg, and M. T. Özsu. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology*, pages 712–729, 2004.

[13] L. Guo, D. Zhang, G. Li, K.-L. Tan, and Z. Bao. Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In *ACM SIGMOD*, page 843–857, 2015.

[14] D. He, S. Wang, X. Zhou, and R. Cheng. GLAD: A grid and labeling framework with scheduling for conflict-aware knn queries. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019.

[15] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, et al. IBM streams processing language: analyzing big data in motion. *IBM Journal of Research and Development*, pages 7–1, 2013.

[16] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Data Engineering, International Conference on*, pages 341–352, 2003.

[17] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. page 804–815. VLDB, 2004.

[18] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. LARS: A location-aware recommender system. In *International Conference on Data Engineering*, pages 450–461, 2012.

[19] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.

[20] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.1. Technical report, 2015.

[22] H. Miao, H. Park, M. Jeon, G. Pekhimenko, et al. StreamBox: Modern stream processing on a multicore machine. In *USENIX ATC 17*, pages 617–629, 2017.

[23] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. *GeoInformatica*, pages 343–365, 2005.

[24] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Transactions on Knowledge and Data Engineering*, pages 789–803, 2007.

[25] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In *USENIX Annual Technical Conference*, pages 493–505, 2016.

[26] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM*, volume 1, pages 7–16, 2011.

[27] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *VLDB*, pages 709–720, 2014.

[28] A. Shahvarani and H.-A. Jacobsen. A hybrid B+-Tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *SIGMOD*, page 1523–1538, 2016.

[29] A. Shahvarani and H.-A. Jacobsen. Parallel index-based stream join on a multicore CPU. In *SIGMOD*, page 2523–2537, 2020.

[30] B. Shen, Y. Zhao, G. Li, W. Zheng, Y. Qin, B. Yuan, and Y. Rao. V-Tree: Efficient kNN search on moving objects with road-network constraints. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 609–620, 2017.

[31] E. Stehle and H.-A. Jacobsen. A memory bandwidth-efficient hybrid radix sort on GPUs. In *SIGMOD*, page 417–432, 2017.

[32] E. Stehle and H.-A. Jacobsen. ParPaRaw: Massively parallel parsing of delimiter-separated raw data. *VLDB*, page 616–628, 2020.

[33] D. Taniar and W. Rahayu. A taxonomy for nearest neighbour queries in spatial databases. *Journal of Computer and System Sciences*, pages 1017 – 1039, 2013.

[34] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Sigmod*, pages 625–636, 2011.

[35] The New York City Taxi and Limousine Commission. TLC trip record data, "https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page", 2021.

[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, et al. Storm@Twitter. In *SIGMOD*, pages 147–156, 2014.

[37] Z. Xu and H.-A. Jacobsen. Adaptive Location Constraint Processing. In *SIGMOD*, pages 581–592, 2007.

[38] Z. Xu and H.-A. Jacobsen. Processing Proximity Relations in Road Networks. In *SIGMOD*, pages 243–254, 2010.

[39] C. Yang, X. Yu, and Y. Liu. Continuous knn join processing for real-time recommendation. In *2014 IEEE International Conference on Data Mining*, pages 640–649, 2014.

[40] H. Yin, Y. Sun, B. Cui, Z. Hu, and L. Chen. LCARS: A location-content-aware recommender system. In *ACM SIGKDD*, page 221–229, 2013.

[41] M. Zaharia, R. S. Xin, P. Wendell, T. Das, et al. Apache Spark: A unified engine for big data processing. *Communication of the ACM*, pages 56–65, 2016.

[42] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *VLDB*, page 516–530, 2019.