Parallelizing Intra-Window Join on Multicores: An Experimental Study

Shuhao Zhang
Singapore University of Technology and Design*

Yancan Mao
National University of Singapore

Jiong He
ByteDance
Singapore

Philipp M. Grulich
Technische Universität Berlin

Steffen Zeuch
German Research Centre for Artificial Intelligence

Bingsheng He
National University of Singapore

Richard T. B. Ma
National University of Singapore

Volker Markl
Technische Universität Berlin, German Research Centre for Artificial Intelligence

ABSTRACT

The *intra-window join* (*IaWJ*), i.e., joining two input streams over a single window, is a core operation in modern stream processing applications. This paper presents the first comprehensive study on parallelizing the *IaWJ* on modern multicore architectures. In particular, we classify *IaWJ* algorithms into *lazy* and *eager* execution approaches. For each approach, there are further design aspects to consider, including different join methods and partitioning schemes, leading to a large design space. Our results show that none of the algorithms always performs the best, and the choice of the most performant algorithm depends on: (i) workload characteristics, (ii) application requirements, and (iii) hardware architectures. Based on the evaluation results, we propose a decision tree that can guide the selection of an appropriate algorithm.

CCS CONCEPTS

• **Information systems → Stream management**; • **Computing methodologies → Shared memory algorithms**; • **Computer systems organization → Multicore architectures**.

KEYWORDS

stream processing; window join; multicores

ACM Reference Format:
Shuhao Zhang, Yancan Mao, Jiong He, Philipp M. Grulich, Steffen Zeuch, Bingsheng He, Richard T. B. Ma, and Volker Markl. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event , China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452793

1 INTRODUCTION

The join of multiple data streams is a common operation that is relevant to many applications, such as online data mining and interactive query processing [4, 21]. To handle infinite streams, the join is typically performed over bounded, discrete subsets of streams (i.e., windows). Such an operation is called a window join and has been adopted in most modern stream processing engines [9, 45]. A large body of prior works focus on joining unbounded overlapping (i.e., sliding) windows, which we denoted as *inter-window join*. For these works, the key concern is to enable incremental computation through continuous window updates while achieving efficient workload distribution [32, 40, 44]. In contrast, the *intra-window join* (*IaWJ*), also called *online join* [14] and *full-history join* [27], has received less attention, despite that it has many practical applications where users are only interested in joining one particular subset of input streams [1, 4, 14, 16, 27]. For example, at Pinterest [33], developers applied the *IaWJ* operation of Flink (called IntervalJoin [1]) to join the activation record per user for a single time window of three days [23].

Fully exploiting current and emerging hardware trends is a notorious challenge [28, 34, 39, 42, 44, 54]. In particular, there are three key challenges (***C1~C3***) to answer the question of how to efficiently parallelize *IaWJ* on multicore architectures. ***C1:*** a significant number of join algorithms have been proposed and can be applied. We classify them into two fundamental *execution approaches*: lazy and eager. The lazy approach first buffers all input tuples of the concerned window from both input streams, and then joins a complete set of tuples. In contrast, the eager approach aggressively joins subsets of input tuples upon the arrival. Furthermore, for each execution approach, there are two further design aspects including *join methods* (i.e., hash- or sort-based) and various *partitioning schemes* (e.g., with or without physical replication), leading to a large design space. ***C2:*** input workloads can vary significantly with different (i) key-skewness, (ii) timestamp-skewness, (iii) tuple arrival rate, (iv) window length, and (v) number of duplicates per key. Meanwhile, applications may target different performance metrics such as throughput [52], latency [22], and progressiveness [13]. These metrics meet divergent requirements and can sometimes conflict with each other. ***C3:*** modern hardware features such as advanced vector extensions (AVX), multicore parallelism, and a complex cache

*Work done while as a postdoc researcher at TU Berlin Germany.

SIGMOD '21, June 20–25, 2021, Virtual Event , China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452793

Table 1: Notations used in this paper

Notations	Description
$x = \{t, k, v\}$	An input tuple x with three attributes
R, S	Two input streams to join
$skew_{key}$	Key skewness (unique or zipf)
$skew_{ts}$	Timestamp skewness (uniform or zipf)
$dupe$	Average number of duplicates per key
v	Input arrival rate (tuples/msec)
w	Window length (msec)

memory subsystem further enlarge the design space. Due to the lack of a thorough study, it is difficult for researchers and practitioners to determine the optimal approach under different conditions.

To the best of our knowledge, this work presents the first comprehensive experimental study on the effectiveness of *IaWJ* algorithms on modern multicores. To this end, we extend an existing benchmark framework [5] by reimplementing eight algorithms and integrating them into the same codebase, eliminating the differences caused by programming languages and compilers to address ***C1***. We propose four representative real-world workloads and one carefully designed synthetic workload, as well as mechanisms to automate the evaluation of three performance metrics to address ***C2***. For ***C3***, we conduct our experiments on a recent multicore processor under different configurations, including altering AVX instructions and a varying number of CPU cores. All source code and data of our benchmark as well as guidelines on how to reproduce our work are publicly available [41].

2 BACKGROUND

We summarize the notations used throughout this paper in Table 1. We define a *tuple* x as triple $x = t, k, v$, where t, k and v are the timestamp, key, and payload of the tuple, respectively. We define the *input stream* (denoted as R or S) as a list of tuples chronologically arriving at the system (e.g., a query processor). We denote the *key skewness* of one input stream (or a subset) as $skew_{key}$ and the *timestamp skewness* as $skew_{ts}$. With a higher $skew_{ts}$, more input tuples are skewed toward the same timestamp. We use *dupe* to denote the average number of duplicates per key in the input stream. The *input arrival rate* (denoted as v) stands for the number of tuples arriving per unit of time from one input stream and w denotes the length of the concerned window to join.

By definition, joining over streams is performed over infinite input tuples. In practice, users typically formulate queries that compute joins over bounded subsets of streams, called *windows* [29]. We adopt a time-based window scheme defined as follows:

Definition 1 ($Window$)**.** *We define a* ***window*** *as an arbitrary time range ($t_1 \sim t_2$) with a length of* w*, i.e.,* $t_2 - t_1 = w$*. For brevity, we henceforth denote a window as* w*.*

Most prior works [32, 40, 44] target the *inter-window join*, which joins streams over infinite sliding (i.e., overlapping) windows. For those works, the primary concern is to efficiently explore *incremental computation* across windows. In contrast, we focus on *IaWJ* that joins streams over a single window [14, 27] regardless of the window type [46] (i.e., sliding, tumbling, or session). Designing efficient *inter-window join* algorithms by taking *IaWJ* as a building block is an exciting topic for further investigation that is beyond the scope of this paper.

Table 2: Summary of studied join algorithms

Name	Approach	Join Method	Partitioning Schemes
`NPJ` [8]	Lazy	Hash	No physical partitioning
`PRJ` [25]	Lazy	Hash	Cache size-aware replication
`MWay` [11]	Lazy	Sort	Equisized range partitioning
`MPass` [5]	Lazy	Sort	Equisized range partitioning
$\texttt{SHJ}^{\texttt{JM}}$ [49]+[14]	Eager	Hash	Content-insensitive stream distribution
$\texttt{SHJ}^{\texttt{JB}}$ [49]+[27]	Eager	Hash	Content-sensitive stream distribution
$\texttt{PMJ}^{\texttt{JM}}$ [13]+[14]	Eager	Sort	Content-insensitive stream distribution
$\texttt{PMJ}^{\texttt{JB}}$ [13]+[27]	Eager	Sort	Content-sensitive stream distribution

Definition 2 (*intra-window join*)**.** *Given input streams R and S and a window w, the* ***intra-window join*** *joins a pair of subsets (i.e., R', S') such that $R' \bowtie S' = \{(r \cup s) | r.key{=}s.key, r.ts \in w, s.ts \in w, r \in R, s \in S\}$, where each result tuple $(r \cup s)$ has a timestamp, key, and value of* $\max(r.ts, s.ts)$*, $r.key$, and $r.value \parallel s.value$, respectively.*

3 STUDIED ALGORITHMS

An important dimension to classify join algorithms is the *execution approach*, i.e., lazy or eager. First, relational join algorithms [5, 6] widely used in conventional databases can be applied to implement the *lazy* join approach. Essentially, they can simply wait for a period of time equal to the window length and then start processing (e.g., constructing the hash table); Second, many specifically designed stream join algorithms [13, 16, 26, 27, 30, 32, 43, 47] can be applied to implement the *eager* join approach. They produce partial matches early and continuously as soon as any input tuples from either input stream arrive. In this work, we studied eight representative join algorithms covering a large design space. We summarize them in Table 2, where each column denotes an important design aspect.

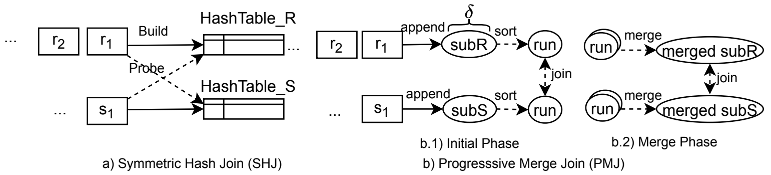
3.1 Lazy Join Approach

In the following, we discuss two hash-based and two sort-based relational join algorithms as the representative lazy algorithms.

No-Partitioning Join (`NPJ`). `NPJ` [8] is a parallel version of the canonical hash join algorithm. Both input relations are divided into equisized portions to be assigned to a number of threads. In the build phase, all threads populate a shared hash table with all tuples of R. After synchronization via a barrier, all threads enter the probe phase, and concurrently find matching join tuples in their assigned portions of S.

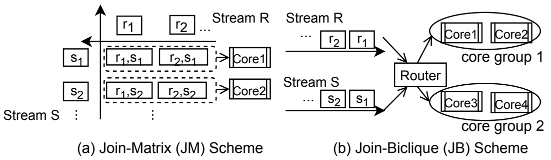
Parallel Radix Join (`PRJ`). `PRJ` [25] subdivides both input relations based on the binary digits of keys (i.e., radix), and physically assigns the resulting sub-relations into individual threads, and thus avoids the hash table being shared among threads in the case of `NPJ`. The goal of the physical relation partitioning is to break at least the smaller input (i.e., tuples from R) into pieces that fit into the caches. Thereafter, it can launch a cache-resident hash join for each partition.

Multi-Way Sort Merge Join (`MWay`). `MWay` [11] is a parallel version of the canonical sort join. The algorithm proceeds as follows: First, the input relations R and S are physically partitioned by key range and equally distributed across CPU sockets. Then, each local partition is sorted using the AVX sorting instructions, and all local partitions are sorted in parallel. Subsequently, multi-way merging is applied to shuffle and merge data among different partitions to obtain a globally sorted copy of R and S. Finally, each thread concurrently evaluates each pair of tuples between NUMA-local sorted subset of inputs using a single-pass merge join.



a) Symmetric Hash Join (SHJ) b) Progresssive Merge Join (PMJ)

Figure 1: Stream join algorithms.



(a) Join-Matrix (JM) Scheme (b) Join-Biclique (JB) Scheme

Figure 2: Stream distribution schemes.

Multi-Pass Sort Merge Join (MPass). The second variant of the sort join is called MPass [5]. It differs from MWay only in the shuffling and merging phases. MWay scales poorly with increasing input size [31] due to the multi-way merge. Instead, MPass applies successive two-way bitonic merging multiple iterations.

3.2 Eager Join Approach

The *eager* algorithm can be constructed by combining a stream join algorithm and a stream distribution scheme. In the following, we discuss two stream join algorithms (i.e., SHJ and PMJ) and two stream distribution schemes (i.e., JM and JB), illustrated in Figure 1 and Figure 2, respectively.

3.2.1 Stream Join Algorithms. When designing stream join algorithms, prior work [13, 16, 26, 30, 43, 47] has historically focused on single-thread execution efficiency while resolving disk I/O issues due to the limitation of hardware resources.

Symmetric Hash Join (SHJ). SHJ is considered to be the first hash-based stream join algorithm [49]. The crux of this approach is to interleave the build and probe processes. Its overall process is illustrated in Figure 1a. It maintains two hash tables, one for each input stream. When the algorithm receives a tuple from R (or S), it inserts the tuple into the hash table of R (or S) and immediately probe the hash table of the opposite stream S (or R). This process continues until all tuples from both input streams have been consumed. Due to its simplicity and its design goal of achieving low processing latency, SHJ has been the default (and often the only available) join algorithm used in many state-of-the-art stream processing engines [36]. However, as we will demonstrate later, there may be other join algorithms that are more suitable.

Progressive Merge Join (PMJ). Dittrich et al. [13] proposed a generic technique called progressive merge join (PMJ) that eliminates the blocking behavior of sort-based join algorithms. The key idea of PMJ is to first read tuples from both input streams until it hits the memory space constraint. The loaded subsets are then sorted. The resulting sorted subsets (called *runs*) are immediately joined with a simple sequential scan. Both *runs* are subsequently stored on disk, and later revisited to merge and produce further matches among different pairs of subsets. We slightly modify PMJ to make it better fit for modern hardware. We illustrate the algorithm in Figure 1b. The key modifications are that we use a parameter δ to control the number of input tuples to be accumulated at each iteration (instead of hitting memory constraint), and all *runs* are subsequently stored in the main memory instead of disk. Many such *runs* may be generated and need to be merged later. Tuning the parameter δ enables the trade-off between the delay of starting the process and the number of *runs* generated. When δ is set to the main memory size, the algorithm falls back to the original PMJ.

3.2.2 Stream Distribution Schemes. To cope with the rapid growth of data rates, researchers have recently proposed to parallelize single-thread stream join algorithms [14, 27]. The key idea is to dynamically distribute input streams among multiple threads. At the same time, each thread launches a stream join algorithm to process the assigned input tuples. Both the distribution and processing are eagerly conducted continuously for every incoming input tuple. In this work, we study two stream distribution schemes: 1) join-matrix (JM) [14] and 2) join-biclique (JB) [27], where the former is content-insensitive, and the latter is content-sensitive.

Join-Matrix (JM). We illustrate JM in Figure 2a. It abstracts the join process between two streams as a matrix, where each cell of the matrix represents a join between a pair of tuples from two input streams [14]. We can partition the matrix and assign a portion to every thread. JM achieves a balanced workload distribution among threads and is robust in the presence of skewness as the workload partitioning is content-insensitive. However, as studied by Lin et al. [27], JM has a major drawback of communication efficiency, as it has to replicate a significant amount of input tuples. For example, r_1 and r_2 are duplicated among threads in Figure 2a.

Join-Biclique (JB). The JB scheme organizes the processing units as a complete bipartite graph, where each side corresponds to an input stream. We illustrate the overall process of JB in Figure 2b. Multiple *core groups* are maintained by the system. Input streams are first passed to a router that decides to which *core group* the tuple is sent. Compared to JM, JB can be tuned towards workload balance and efficiency by adjusting the routing strategy. We will discuss the effect of the tuning parameters in Section 5.5.

4 METHODOLOGY

In this section, we first introduce the examined performance metrics. Then, we present our benchmark suite implementation.

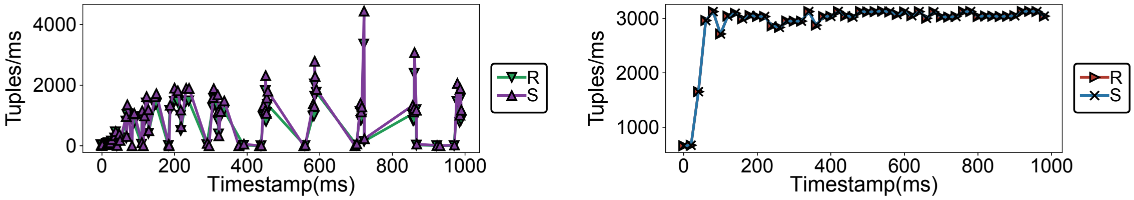
4.1 Performance Metrics

Throughout this study, we focus on three important performance metrics of streaming applications. First, *throughput* represents the overall processing efficiency. It is defined by the number of input tuples processed per unit of time. Second, *latency* describes the difference between the time when a match is generated and its last corresponding input (R or S) arrives. Following previous work [22], we measure quantile worst-case latency (e.g., 95^{th}). Lastly, *progressiveness* is a widely used performance metric to examine how algorithms make progress [47, 49] in delivering partial results (e.g., the top 50%). It is usually represented as the cumulative percentile of matches as a function of elapsed time.

Optimizing algorithms for all three performance metrics is difficult. On the one hand, these metrics are all useful in different use cases but sometimes conflict with each other. On the other hand, existing join algorithms are often designed to optimize only one or two performance metrics. For example, relational join algorithms [5] are primarily designed to optimize throughput. In

Table 3: Statistics of four real-world workloads (w=1000ms)

	Arrival rate (tuples/ms)	Key duplicates	Key skewness (Zipf)	Number of tuples
Stock	v_R=61, v_S=77	$dupe(R)$≈67.7, $dupe(S)$≈78.5	$skew_{key}(R)$=0.112, $skew_{key}(S)$=0.158	$\lvert R\rvert$ ($\lvert S\rvert$) = $v_{R(S)} \cdot w$
Rovio	$v_R≈3 \cdot 10^3$, $v_S≈3 \cdot 10^3$	$dupe(R)$=$dupe(S)$=17960.0	$skew_{key}(R)$=0.042, $skew_{key}(S)$=0.042	$\lvert R\rvert$ ($\lvert S\rvert$) = $v_{R(S)} \cdot w$
YSB	$v_R=\infty$, $v_S≈10^4$	$dupe(R)$=1, $dupe(S)=10^5$	$skew_{key}(R)$=0.033, $skew_{key}(S)$=0.032	$\lvert R\rvert=10^3$, $\lvert S\rvert=v_S \cdot w$
DEBS	$v_R=\infty$, $v_S=\infty$	$dupe(R)$≈172.6, $dupe(S)$≈111.5	$skew_{key}(R)$=0.003, $skew_{key}(S)$=0.011	$\lvert R\rvert=10^6$, $\lvert S\rvert=10^6$



(a) Traded & Quote data of Stock

(b) Ad & Purchases data of Rovio

Figure 3: Time distribution of Stock and Rovio. Other uniform arrival datasets are shown as horizontal lines and omitted.

contrast, stream join algorithms are typically designed to optimize latency [21] and/or progressiveness [13, 49].

4.2 Benchmark Suite

In this section, we introduce our benchmark suite covering selected workloads and implementations.

4.2.1 Benchmark Workloads. In the following, we describe each workload including its application scenario and input setup. For each workload, instead of considering the whole query, we only execute and measure the join process.

Stock: One common stock analysis task is to get the turnover rates of stocks by calculating the trade ratio of each stock every period of time. The input data are a stream of quotes and a traded stream that contains traded results of matched quotes. We use a real-world stock exchange dataset [2] for this workload. The query joins the traded stream (R) and the quotes stream (S) over the same stock id within the same period of time (i.e., window).

Rovio: Rovio [37] continuously monitors the user actions of a given game to ensure that their services work as expected [22]. One use case is to correlate advertisements with their revenue, where the input data can be a purchase stream that contains tuples of purchased gem packs and an advertisement stream which contains a stream of proposals of gem packs to users. We use the dataset from Karimov et al. [22] for this workload. The query joins the advertisements stream (R) and the purchase stream (S) over the user id and advertisement id within the same window.

YSB: Yahoo Stream Benchmark (YSB) [12] describes a simple job that identifies the campaigns of advertisement events and stores a window count of relevant events per campaign. The input data is a campaigns table that contains 1000 advertising campaigns and an advertisement stream that contains a number of advertisements for each campaign. We use the dataset generator from Chintapalli et al. [12] for this workload. The query joins the campaigns table (R) and the advertisements stream (S) over the campaign id.

DEBS: A common analytic query of social networks is to find the number of posts and comments created by users. We use the social network dataset published by the DEBS'2016 Grand Challenge [15], which simulates posts and comments on a social network application. The input data to the query are the posts (R) and comments (S) created by users in the forum. Both inputs are stored at rest and can be viewed as an input stream having a window length of zero and arrival rates of infinite. The query joins *all* tuples of the comments and the posts over the same user id.

We summarize the properties of the four real-world workloads [2, 12, 15, 22] in Table 3. Our selected datasets cover a wide range of workload features. i) **Stock** and **Rovio** have relatively low arrival rates (v); **YSB** has high arrival rates; **DEBS** represents data at rest and the arrival rate is infinity. ii) Figure 3 shows the timestamp distribution of **Stock** and **Rovio**. **Stock** contains obvious event "spikes", where many tuples arrive at the same time slot. On the other hand, **Rovio** has a relatively stable event arrival pattern, where the arriving tuple rate remains roughly the same at most timestamps. Stream S of **YSB** has a uniform time distribution ($skew_{ts}$=0). Stream R of **YSB** and both input streams of **DEBS** are stored at rest, i.e., all tuples arrive instantly. iii) **Stock** has a more skewed workload (i.e., $skew_{key}$ is higher), while others are less skewed. iv) Depending on the use cases, the latency requirement for stream processing can vary from tens of milliseconds to a few seconds [48]. To represent real-world use cases, we set the window length (w) to 1 second for all datasets and study the impact of different window lengths in Section 5.4. v) **Rovio** and **DEBS** have relatively high key duplication in both streams; **YSB** has a low key duplication in R but high in S; and **Stock** has relatively low key duplication in both streams.

Synthetic Workload. The real-world dataset may not cover all situations and and it is not possible to tune its workload characteristics to observe the impacts. To comprehensively evaluate the impact of varying workload properties, we further evaluate a synthetic workload based on the work from Kim et al. [25]. To differentiate with the other four real-world workloads, we refer to this synthetic workload as **Micro**.

4.2.2 Implementations. We discuss three key components of our benchmark suite implementation: the algorithm implementation, the dataset structure design, and the profiling methods.

Algorithm Implementation. The source code of the lazy join algorithms NPJ, PRJ, MWay, MPass are based on the benchmark from Balkesen et al. [5, 6]. The eager algorithms $\mathrm{SHJ}^{\mathrm{JM}}$, $\mathrm{SHJ}^{\mathrm{JB}}$, $\mathrm{PMJ}^{\mathrm{JM}}$, $\mathrm{PMJ}^{\mathrm{JB}}$ are implemented based on the corresponding papers. We reuse the available functions provided by the benchmark from Balkesen et al. [5, 6] as much as possible. Specifically, we use the original *avxsort* function used in MWay and MPass to implement

PMJ's sort and the implementation of *bucket chain hash table* used in PRJ to implement the hash table of SHJ. The original benchmark suite [5, 6] is only to evaluate relational join algorithms on static datasets. We modify it to support *intra-window join* on both static and streaming datasets. We let the lazy algorithms accumulate all input tuples to arrive before start processing. For the eager algorithms, every thread maintains a timer to record its elapsed time since it starts, and alternatively reads from its assigned subsets of input streams (Section 3.2.2) during execution. When tuples from one input (R or S) are not available (i.e., the tuple has a larger arrival timestamp than the thread's elapsed time), the thread attempts to read from the other input stream. Hence, the eager algorithms may still stall if they consume tuples faster than tuple arrival.

Dataset Structure. We follow the same dataset structure of previous works [5, 6]. In particular, they all assume a column-oriented storage model, and joins are assumed to be evaluated over a narrow <$key, payload$> tuple configuration. To make use of vectorized instructions, we assume that each tuple has a width of 64 bits; thus, key and payload are four bytes each. To eliminate the impact of network transmission overhead, the input datasets are first populated (synthetic datasets) or loaded (real datasets) in memory. Then, we assign each tuple a timestamp (starting from 0) to reflect its actual arrival time to the system, and tuples are time ordered. The timestamp is stored as the payload of each tuple.

Profiling Methods. Note that there is additional overhead of checking thread's elapsed timestamp during execution for the eager approach. To minimize this overhead, we use *Read Time-Stamp Counter* (RDTSC) [18] instructions for time measurement, and we observe that such overhead accounts for less than 5% of overall execution time. During processing, every thread records the current timestamp whenever a match is generated by it. When the program finishes, we merge and sort timestamps of matches from all threads to examine the overall progress of the algorithm. Throughput is measured as the total number of inputs divided by the timestamp of the last match. Latency is measured by subtracting the timestamp of a match by the larger timestamp of its corresponding input tuple. Besides performance metrics, we also use *Intel PCM* [20] and *Perf* [35] to gather architectural statistics [19] of processors.

5 EVALUATION

The main objective of the experiments is to comprehend the behaviour of different *IaWJ* algorithms on modern multicores. Specifically, we seek answers to the following questions.

Q1: How do different join algorithms perform on joining real-world workloads with both static and streaming inputs?

Q2: Are there any common performance issues among different algorithms when running on modern multicore processors?

Q3: What are the impacts of workload characteristics on the three performance metrics?

Q4: For each algorithm, how much do the algorithmic parameters affect its performance, e.g., overall execution cost?

Q5: How different join algorithms interact with modern hardware architectures?

Q6: How evaluation results change with or without utilizing SIMD instructions in join algorithms? Do the algorithms scale linearly on multicore processors?

Table 4: Specification of our evaluation platform

Component	Description
Processor (w/o HT)	Intel(R) Xeon(R) Gold 6126 CPU, 2 (socket) * 12 * 2.6GHz
L3 cache size	19MB
Memory	64GB, DDR4 2666 MHz
OS & Compiler	Linux 4.15.0, compile with g++ O3

In the following, we present a summary of our experimental results in Section 5.1. In Section 5.2, we compare the performance of all algorithms using four real-world workloads to answer **Q1**. We provide a detailed execution time breakdown in Section 5.3 to answer **Q2**. In Section 5.4, we evaluate the impact of varying workload configurations using the synthetic workload to answer **Q3**. To answer **Q4**, we perform sensitivity studies on the algorithm parameters in Section 5.5. Finally, we profile hardware counters and evaluate all algorithms with altering AVX instructions and a varying numbers of CPU cores in Section 5.6 to answer **Q5** and **Q6**.

All experiments are carried out on a Intel Xeon Gold 6126 processor. Table 4 shows the detailed specification of the hardware and software used in our experiments. To exclude the impact of NUMA, we only use one socket in our experiment. Note that, MWay and MPass require the number of threads to be a power of 2. For fair comparison, we use 1 to 8 threads in all of our experiments.

5.1 Key Findings

- Our experimental results show that both lazy and eager approaches are relevant. However, no one algorithm can outperform others in all cases (**Q1**). Each algorithm can outperform others in handling specific workloads in terms of one or more performance metrics. Notably, contrary to the common belief [27], our results indicate that relational join algorithms (i.e., lazy) outperform specifically designed stream join algorithms (i.e., eager) on most workloads. Depending on the workload, they achieve up to 5x higher throughput, 10x lower latency, and even much better progressiveness (see Section 5.2).
- In Section 5.3.1, we show that the eager algorithms incur more cache misses during partitioning and probe processes, resulting in higher execution cost (**Q2**). Sort-based join algorithms reduce cache misses in handling high key duplication workloads and achieve better performance for both lazy and eager algorithms. Surprisingly, compared to JM, the JB scheme involves higher partitioning cost due to status maintenance (see Section 5.3.3).
- The eager algorithms achieve better performance when one input stream has a low arrival rate (**Q3**). In particular, SHJ$^{\text{JM}}$ is able to deliver output quickly if the input rate of the data stream varies with spikes (see Section 5.4). However, when both input streams have high arrival rates, the lazy algorithms perform better in all performance metrics.
- The lazy algorithms are more sensitive to tuning parameters, as we observe a significant difference of execution costs between a good and bad configuration (see Section 5.5). In contrast, large constant overhead makes eager algorithms less sensitive to tuning parameters (**Q4**).
- In Section 5.6, we show that eager algorithms are more *Core Bound* [19] (**Q5**). The frequent function calls to pulling data from both input streams overloads the out-of-order execution units. The eager algorithms are also more memory bounded

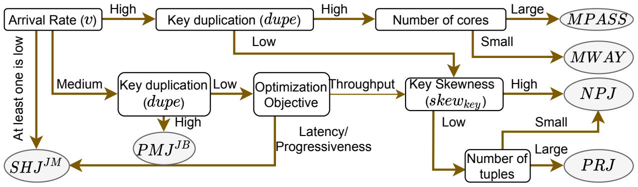


Figure 4: Decision tree for picking an appropriate algorithm. The root node of the tree is the arrival rate node.

due to severe cache misses (in particular L1D-Cache miss). Furthermore, they consume more memory spaces due to additional intermediate results.

- All algorithms scale almost linearly with the number of cores for intensive workloads as there are no major synchronization barriers. AVX instruction set does bring performance improvement over both lazy and eager algorithms, but its improvement for eager algorithms is marginal (**Q6**).

We summarize the main findings of our analysis in a decision tree to guide readers through our results and to aid practitioners in selecting a suitable algorithm to parallelize *IaWJ* on multicore processors (Figure 4). Note that the qualitative remarks of high, medium, and low are relative and the quantitative value depends on actual hardware and workloads. First, we recommend the lazy approach when arrival rates are high. When the key duplication is also high, MPass and MWay are better options and MPass scales better with a large core counts. When the key duplication is low, NPJ and PRJ are more effective, and PRJ performs better when the key distribution is low and the number of tuples to join is large. Second, when the arrival rate is medium, PMJ^{JB} performs best in terms of all three performance metrics for handling high key duplication workload, while SHJ^{JM} achieves lower latency and better progressiveness for handling workloads with low key duplication. If throughput is the key target, we recommend the lazy approach when the input arrival rate is medium and the key duplication is low. Finally, we recommend SHJ^{JM} whenever one input stream has low arrival rate, as it is able to eagerly utilize hardware resources with low overhead.

5.2 Performance Comparison

In this section, we compare algorithms on processing real-world workloads in terms of throughput, latency, and progressiveness. Each algorithm is tuned to its optimal configuration.

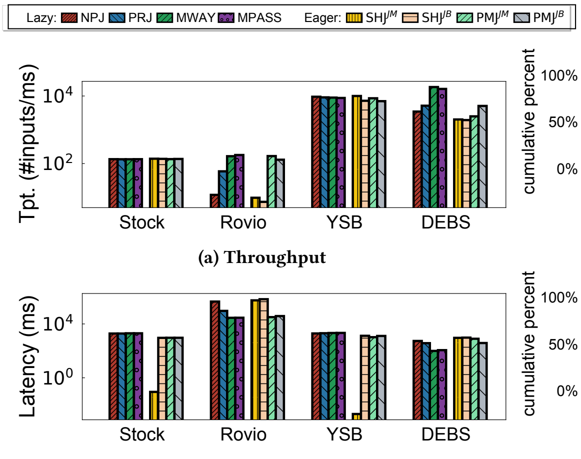
Throughput and Latency. The overall performance comparison results are shown in Figure 5. There are two major observations. First, despite the great differences among workloads, the lazy algorithms always achieve better or at least comparable throughput. The results clearly reflect their better execution efficiency. As expected, the throughput is especially higher on static datasets (i.e., **DEBS**), and the throughput difference is up to 5x. More interestingly, they even achieve higher throughput for joining data in motion, e.g., **Rovio** and **YSB**. When the input arrival rate is low (e.g., **Stock**), hardware resources are underutilized (i.e., up to 6.6% CPU and 0.55% memory bandwidth utilization), and all algorithms show a similar throughput. In handling such workloads, users may optimize the other two performance metrics by selecting appropriate algorithms without sacrificing throughput. Second, the eager algorithms achieve smaller processing latency for **Stock** and **YSB**, although the throughput is similar or even worse compared to the lazy algorithms. This matches the previous findings [21], as those algorithms are specifically designed to achieve low processing latency. However, the results also show that the lazy algorithms achieve similar or even lower latency on other workloads, e.g., **DEBS**. This is because processing latency is also highly correlated to execution efficiency. If input tuples are queuing up as the system is not able to consume them instantly, processing latency increases due to the increased waiting time [22]. Our results demonstrate the better execution efficiency of the lazy algorithms and clearly indicate a false claim that specific stream join algorithms are always better suited when handling data streams [27]. Nevertheless, there is no single winner among all algorithms. For example, sort-based algorithms achieve higher throughput for handling **Rovio** and **DEBS**, while hash-based ones are better suited for **Stock** and **YSB**.

Progressiveness. Figure 6 compares how different algorithms make progress and the results show that which approach makes faster progress depends heavily on the workloads. The eager approach is able to generate matches without waiting for all inputs. For example, SHJ^{JM} is able to deliver the first 50% matches of handling **Stock** in 674ms, which is around 1.5x faster than the fastest lazy alternative (i.e., 1014ms by NPJ). However, a lazy algorithm can quickly finish the overall process and it can surpass eager algorithms early. For example, MPass is able to output 50% of total matches in 11699ms when handling **Rovio**, while the fastest eager algorithm PMJ^{JM} produces only 28% of total matches using the same amount of time. The common wisdom [13, 27, 49] that stream join algorithms always make faster progress for handling streaming data is obviously misleading. Our results also indicate an interesting future research area to explore how to orchestrate both approaches to achieve optimal progressiveness at all time.

5.3 Execution Time Breakdown

In this section, we report how much time is spent in the processing of each input among different algorithms. We divide it into six phases: i) **Wait** is the time spent waiting for inputs to arrive. The lazy algorithms wait until the very last tuple of the concerned window arrives before the join starts. Thus, it's *wait* equals window length. The eager algorithms also wait if the system is underutilized (i.e., input arrive rate is lower than the system processing rate). ii) **Partition** is the time spent partitioning workloads among threads. iii) **Build/Sort** is the time spent due to hash table construction or tuple sorting in hash or sort-based algorithms, respectively. iv) **Merge** is the time spent merging tuples, which is only present in sort-based join algorithms. v) **Probe** is the time actually spent on matching tuples. It refers to either the probe time in hash-based or the match time in sort-based join algorithms. vi) **Others** refer to all remaining overheads such as context switching.

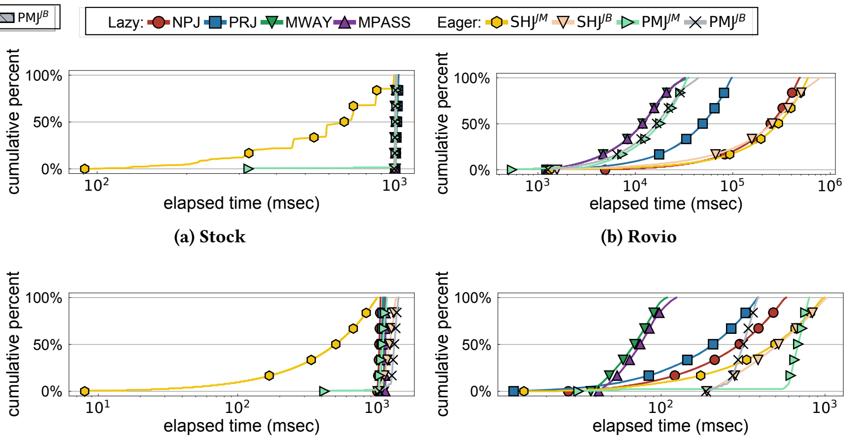
5.3.1 Overall Comparison. The results of the time breakdown are shown in Figure 7. There are two key observations when we compare all algorithms. First, it confirms that all algorithms spend most of the time on *wait* for processing **Stock**, which has a small arrival rate, and all CPU cores are mostly idle. Second, if we exclude the *wait* cost, we can see that the eager algorithms generally involve higher cost per input tuple. In particular, execution time is mostly spent on either probing or partitioning.



(a) Throughput

(b) 95th Latency

Figure 5: Throughput and latency comparison.



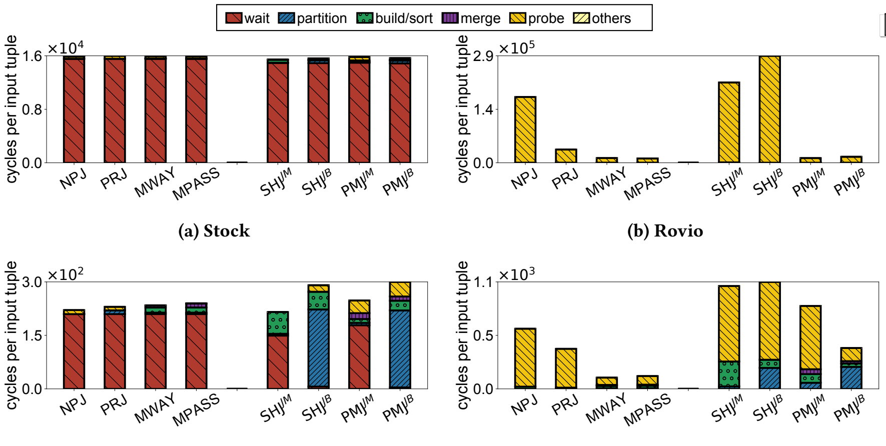
(a) Stock

(b) Rovio

(c) YSB

(d) DEBS

Figure 6: Progressiveness comparison.



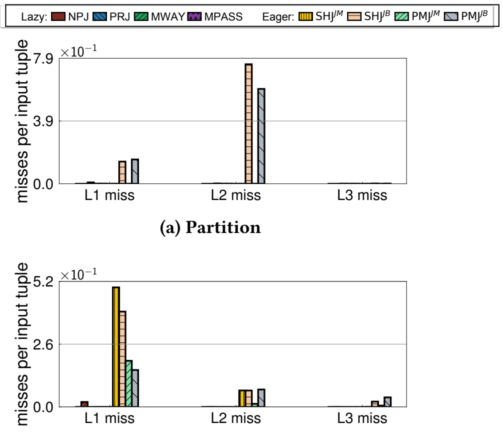
(a) Stock

(b) Rovio

(c) YSB

(d) DEBS

Figure 7: Execution time breakdown.



(a) Partition

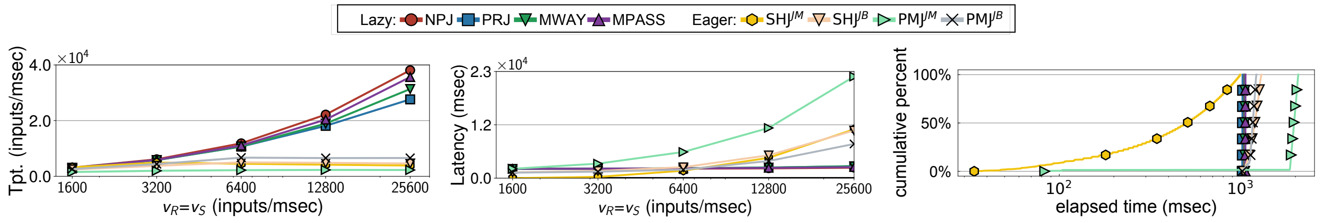
(b) Probe

Figure 8: Cache efficiency profiling (YSB).

We further take the **YSB** as an example to examine the cache efficiency of different algorithms. Figure 8 shows that the inefficiency of eager algorithms is caused by significant cache misses. There are different cache access behaviours during different phases. First, SHJ^{JB} and PMJ^{JB} experience higher cache miss at L1 and L2 cache during the *partitioning* phase. This is mainly caused by their content-sensitive partitioning scheme (i.e., JB). Specifically, each thread accesses tuples according to their primary keys resulting in random memory accesses, and the footprint between consecutive access exceeds L2 cache but smaller than L3 cache. Second, all eager algorithms experience significant L1 cache misses during the *probe* phase. This is due to their frequent interleaving access, where each thread aggressively works on any available tuples from either input stream resulting in cache trashing.

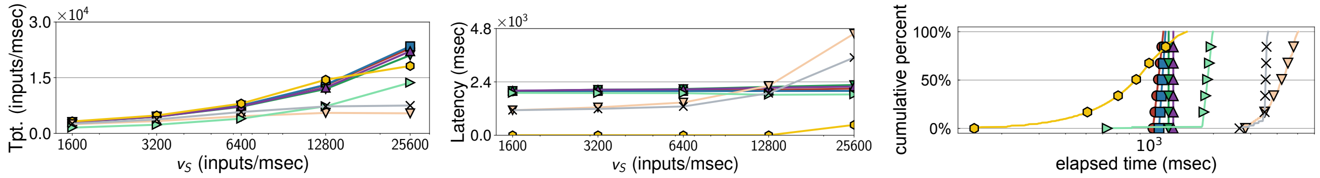
5.3.2 Lazy Approach Comparison. If we compare the lazy join algorithms (i.e., NPJ, PRJ, MWay, and MPass), we can see that hash-based algorithms involve much higher costs in *probe* when handling **Rovio** and **DEBS**. We find that these datasets have many duplicate keys in both streams (see Table 3). PRJ uses a *bucket-chain* design, where tuples of the same key will be inserted into the same bucket until the bucket is full and a new bucket is created and appended to form a list. When handling high key duplication workloads, each thread needs to walk through a long list to perform the match. NPJ involves high cost for handling high key duplication workloads because of the higher chance of access conflict when the same hash bucket is concurrently visited by multiple threads. These issues are not involved in sort-based algorithms. However, the execution cost of sort-based algorithms is slightly higher than the hash-based algorithms in handling **YSB**, where keys of tuples from R are unique (see Table 3). Our results show that, in such case, there is still a higher *sort* cost compared to *build* cost although AVX instruction sets can accelerate the sorting process, which reaffirms the analysis of prior work by Balkesen et al. [5].

5.3.3 Eager Approach Comparison. Finally, we compare the eager algorithms (i.e., SHJ^{JM}, SHJ^{JB}, PMJ^{JM}, and PMJ^{JB}). We have similar observations of analysing the lazy approach that sort-based algorithms outperform hash-based ones in handling **Rovio** and **DEBS** (those with relatively high key duplications in both input streams). If we compare the eager algorithms with different partitioning schemes, we can see that algorithms with the JB



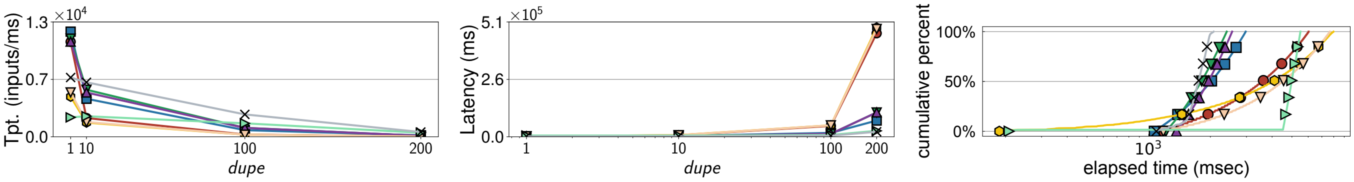
(a) Throughput **(b) 95th Latency** **(c) Progressiviness (v_R = v_S =1600) tuples/msec**

Figure 9: Results of varying input arrival rates.



(a) Throughput **(b) 95th Latency** **(c) Progressiveness (v_S =25600 tuples/msec)**

Figure 10: Impacts of relative arrival rates (v_R = 1600 tuples/msec).



(a) Throughput **(b) 95th Latency** **(c) Progressiveness ($dupe$=100)**

Figure 11: Impacts of key duplication (v = 6400).

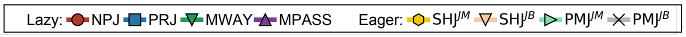
scheme surprisingly involve significant partitioning overhead. We find that this is mainly due to the status maintenance overhead. Specifically, after each tuple is dispatched, the system needs to record the dispatch results for future reference. In the original work by Lin et al. [27], this overhead is negligible because they use Storm [45] as their target test platform, which is known to poorly utilize modern multicore processors [53].

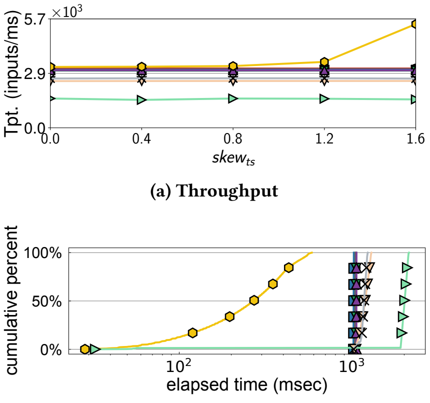
5.4 Workload Sensitivity Study

Workload characteristics of data streams being joined vary significantly among applications. On the one hand, the input data characteristic from the same input stream may vary depending on which subsets of streams are joined and also on the window length. On the other hand, even if data sources (e.g., sensors) may generate data streams (almost) constantly, the actual deployment of stream processing at various infrastructure layers [51] leads to different workloads in the system. In the following, we evaluate the impact of workload characteristics quantitatively. We use **Micro** in this study due to its simplicity and flexibility. We tune the parameters to change the features of the generated datasets.

Impact of Arrival Rate (v). Figure 9 illustrates the impact of the arrival rate of both input streams, ranging from 1600 to 25600 tuples/msec. We fix the window length to 1 second and generate the datasets with a unique key set and uniform arrival distribution. There are two key observations: First, when the arrival rate is low (e.g., v=1600), all join algorithms achieve similar throughput as the CPUs are underutilized, but SHJ^{JM} delivers results constantly faster and guarantees much lower processing latency. Note that the progressive curve in Figure 9c reflects that SHJ^{JM} can instantly process every input tuple once it arrives and can deliver results almost immediately. This result matches the case of handling the **Stock** workload as discussed previously. Second, with an increasing arrival rate, the lazy algorithms can gradually improve the throughput, while the eager algorithms deliver similar or even worse throughput. When the arrival rate is medium (e.g., v=12800), the eager algorithms achieve lower processing latency and better progressiveness, although the throughput is lower compared to lazy algorithms. Due to the lack of complete knowledge of the entire data streams, the eager algorithms miss the opportunities to optimize the processing in a holistic manner and thus will inevitably incur performance penalty such as cache misses. As a result, when the arrival rate is high (e.g., v=25600), the eager algorithms are not able to consume input tuples immediately and perform worse in all performance metrics.

Impact of Relative Arrival Rate (v_R:v_S). We also evaluate the impact of varying relative arrival rates of two input streams. In this experiment, we keep v_R at 1600 tuples/msec and vary v_S from 1600 to 25600 tuples/msec. The evaluation results are shown in Figure 10. We see that SHJ^{JM} can constantly perform best in all three performance metrics with different values of v_S. This is because, as one input stream arrives slowly, SHJ^{JM} continuously reads input tuples from the other stream. This reduces the chance of reading from two input streams interleaved and effectively reduces random memory access. Processing latency remains constant in most algorithms, as matches are mostly triggered by tuples from

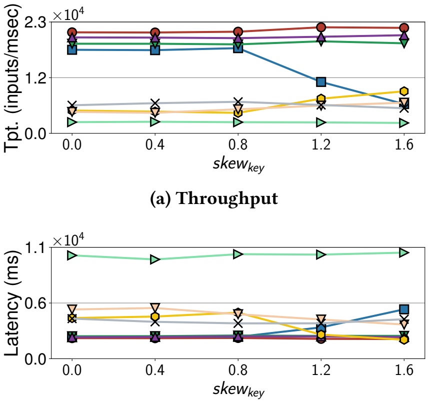




(a) Throughput

(b) Progressiveness ($skew_{ts}$ =1.6)

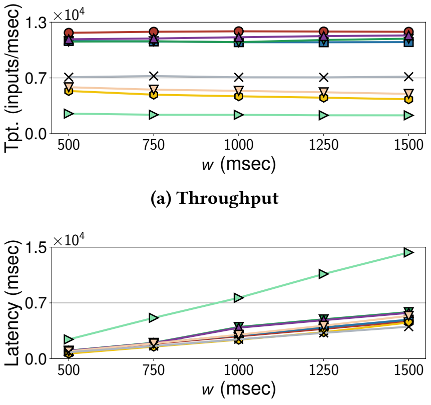
Figure 12: Impacts of arrival skewness (v = 1600).



(a) Throughput

(b) 95th Latency

Figure 13: Impacts of key skewness (v = 12800).



(a) Throughput

(b) 95th Latency

Figure 14: Impacts of window length (v = 12800).

R, which arrive slowly. The latency of PMJ^{JB} and SHJ^{JB} increases significantly when they are not able to keep up with the increasing aggregated arrival rates from both input streams.

Impact of Key Duplication (*dupe*). We now evaluate the impact of duplicate keys in the input streams. We set the input arrival rate of both input streams to 6400 tuples/msec and set the window length to 1000 msec. We then vary the duplication of each key from 1 to 100 times. Thus, the total number of matches become 1 to 100 times larger. The results in Figure 11 show that the sort-based join algorithms outperform hash-based ones in terms of all three performance metrics for both eager and lazy approaches when the *dupe* is greater than 10 and PMJ^{JB} outperforms all other algorithms when the *dupe* is greater than 100. Our further investigation reveals that performance improvement mainly come from two aspects. First, when multiple tuples of the same key are sorted, they are allocated sequentially and cache-aligned in memory. This significantly increases memory bandwidth when they are later revisited during the matching phase. Second, during matching, the cache reuses increase in sort-based algorithms, as each input tuple produces multiple matches due to key duplication. Those results refresh the understanding of the applicability of those sort-based join algorithms, which are not captured in prior work [5].

Impact of Arrival Skewness ($skew_{ts}$). To quantify the impacts of the arrival skewness, we adopt a Zipf-distribution generator to generate tuples. We set the arrival rate to 1600 tuples/msec and fix the window length to 1000 msec for both input streams such that the total number of input tuples remains the same. We examine the case where more tuples bear the same timestamps as in the *early* tuples of input streams with increasing $skew_{ts}$. The results are shown in Figure 12. We omit the results of processing latency as it remains almost unchanged in all algorithms with varying arrival distributions because of the low input arrival rates. Among all algorithms, only SHJ^{JM} is sensitive to varying arrival distributions. In particular, the throughput gradually increases when $skew_{ts}$ exceeds 1.2 and achieves the highest throughput at 1.6. The reason is that it can utilize the hardware resources as early as possible. This is reaffirmed by the progressive evaluation results in Figure 12b.

Impact of Key Skewness ($skew_{key}$). We set input arrival rates of both input streams to be 12800 tuples/msec, the window length to be 1000 msec and a uniform arrival distribution. The results are shown in Figure 13. We can see that only PRJ is less tolerant to key skewness. PRJ relies on recursively repartitioning the input tuples from R until a single partition can fit into the L1 cache. Subsequently, only one thread will work on a single partition. Due to the high key skewness, PRJ underutilizes the hardware resources as only a few partitions are created, and few threads are concurrently running. SHJ^{JM} becomes even better with skewed input due to the better cache behaviour as the same key is repeatably revisited.

Impact of Window Length (w). The window length defines the time range of each substream to join. We set the input arrival rate of both input streams to 12800 tuples/msec and vary the window length from 500 to 2500 msec. The throughput results are shown in Figure 14a. We can see that the throughput of all algorithms almost remains similar with increasing w indicating that the amortized execution cost per input tuple is not impacted significantly by w. As the throughput remains constant, the increasing processing latency shown in Figure 14bb indicates that more input tuples are queuing up with larger values of w. This also explains the slight decrease in throughput of the eager algorithms. When there are increasingly more input tuples arriving at the system and being queued up, the footprint between invocation of the same tuple also increases resulting in more cache misses.

5.5 Impact of Algorithm Configurations

In this section, we study the impact of tuning knobs including (i) sorting step size δ, (ii) group size g of the JB scheme, (iii) the impact of physical partitioning during workload distribution, and (iv) number of radix bit $\#r$. We use **Micro** to conduct this set of experiments and assume that all tuples are instantly available to be processed to eliminate the impact of *wait*. To facilitate the comparisons with existing results [5], the total numbers of tuples from R and S are set to $128 \cdot 10^5$ and $128 \cdot 10^6$, respectively.

Varying sorting step size (δ). In PMJ, sorting step size controls the portion of tuples to acquire before the system starts sorting (and

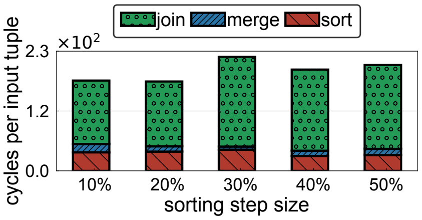
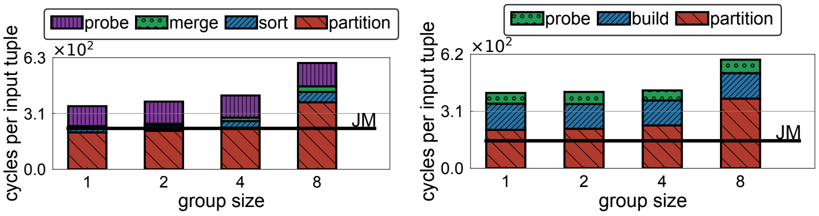


Figure 15: Impact of sorting step size (δ) of PMJ.



(a) $\texttt{PMJ}^{\texttt{JB}}$

(b) $\texttt{SHJ}^{\texttt{JB}}$

Figure 16: Impact of group size (g).

subsequently joining). Figure 15 shows a nontrivial relationship between overall processing cost and sorting step size. A small δ allows the system to wait for a shorter period of time before generating any matches but it may bring more overhead including more context switching and more subsets to merge. In contrast, a large δ essentially defeats the purpose of eagerly processing input streams. A suitable δ (20% in our experiment) brings the optimal performance of both generating preliminary results early and achieving higher overall throughput. For the processing latency, our experimental results show that it does not change on varying δ. This is because the processing latency in worst-case scenario is only affected by matches that are generated mostly in the merge phase. For simplicity, we omit further details of latency and progressiveness evaluation of this experiment.

Varying group size (g). Group size is a key parameter of the JB partitioning scheme. When g=1, JB scheme becomes a strictly hash-partitioning scheme where input streams are hash partitioned among threads by their primary keys. When g equals the number of threads such that there is only a single *core group*, JB becomes the JM scheme and we assume R is replicated while it still partitions S among threads. Hence, the workload increases in each thread with an increasing g. We evaluate the impact of varying g by using both PMJ and SHJ. Figure 16 shows the evaluation results. The horizontal line shows the performance when the JM scheme is applied instead. The evaluation results confirm our analysis of the JB scheme where increasing group size also increases workloads. However, the JM scheme always achieves better performance due to the significant partitioning overhead in JB scheme making the tuning of JB less useful. Note that this contrasts with the observation made in prior work [27] due to the significant difference in the testing environment.

Impact of Physical Partitioning. During the workload partitioning phase of the eager algorithms, we can either pass the pointer or value of each tuple to threads. Passing pointers is faster, but it may lead to potential overhead in subsequent phases. To evaluate its performance impact, we use $\texttt{SHJ}^{\texttt{JM}}$ as an example in this experiment. Figure 17 shows a clear tradeoff among the cost of different phases. On the one hand, it is expected that distributing the value of tuple (w/ partitioning) incurs more *partitioning* cost. On the other hand, physical partitioning results in better cache behaviour during build and merge phases. In our experiment, we observe that both configurations lead to similar overall performance over our selected benchmark workloads. Hence, we simply apply the configuration of passing pointers in all other experiments. In future work, we plan to extend our study to include more workloads and hardware platforms to better comprehend when physical partitioning should be applied in the eager algorithms.

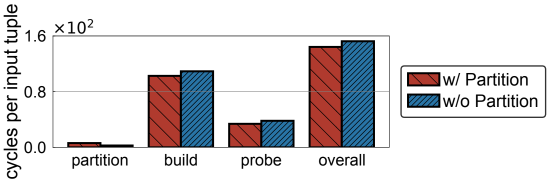


Figure 17: Impact of physical partitioning of $\texttt{SHJ}^{\texttt{JM}}$.

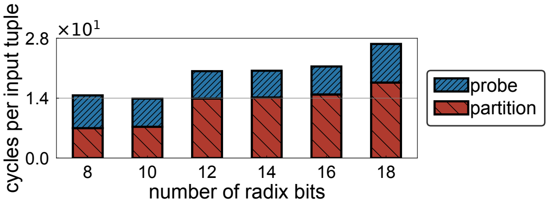


Figure 18: Impact of number of radix bits (#r) of PRJ.

Varying number of radix bits (#r). The number of radix bits is the most important parameter of PRJ, which trades off *partitioning* cost and *probe* cost [6]. We vary #r from 8 to 18 and the results are shown in Figure 18. Our experimental results confirm the observations made in previous work [6] and we experimentally determine the suitable value of #r to be 10 for PRJ in our machine.

5.6 Impact of Modern Hardware

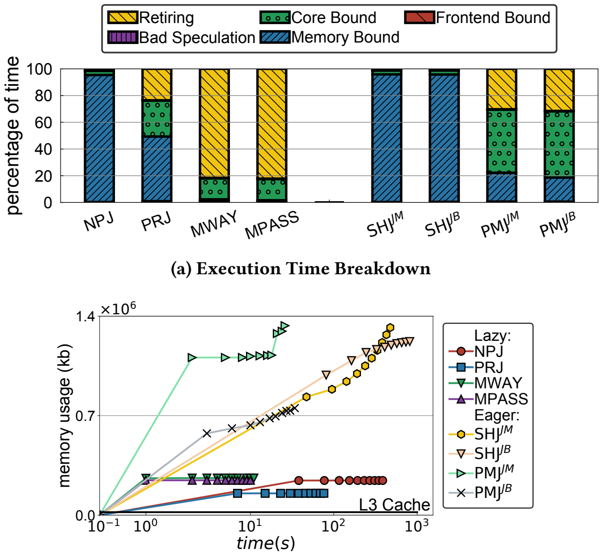
In this section, we first analysis how algorithms interact with modern multicore processors. Then, we show how evaluation results may change under various hardware configurations.

Micro-architectural Analysis. We take **Rovio** as an example to show the breakdown of the execution time according to the Intel Manual [19]. Figure 19a compares the time breakdown of different join algorithms. We measure the hardware performance counters during the algorithm execution, and compute the top-down metrics [50]. We have three major observations. First, the breakdown results reaffirm our previous analysis that sorting-based join algorithms work more efficiently for **Rovio** because of its high key duplication feature. In particular, MWay and MPass show high instruction retiring rate and negligible *Core Bound* and *Memory Bound*. NPJ is more memory bound than PRJ. The reasons are two folds: i) NPJ involves high L1 and L2 cache miss overhead due to the access conflict of the shared hash table; ii) the size of the shared hash table in NPJ exceeds L3 size significantly; thus, resulting in significant L3 miss overhead. Due to the same reason, a high

L3 cache miss issue is also observed in SHJ^{JM} and SHJ^{JB} when handling the **Rovio** workload. Second, although PMJ is also a sort-based algorithm, it shows a much higher *Core Bound* compared to MWay and MPass. This is because of the frequent function calls as PMJ repeatedly acquires new tuples from input streams to process and thus overloads the out-of-order execution units. Such an eager nature also leads to a higher *Memory Bound* in all eager algorithms due to cache thrashing. Third, another interesting observation is that the JB scheme leads to a higher *Core Bound* than JM. We observe that this is due to the additional shuffle phase in the JB scheme resulting in a more complex instruction flow involving more dependencies.

Figure 19b illustrates memory consumption of all algorithms over time. There are two major takeaways. First, we can see that the eager algorithms consume more memory compared to lazy ones. The reason is that although the input tuple may be consumed immediately, it cannot be eliminated as it may be revisited later due to the pointer-based passing mechanism (see Section 5.5). Furthermore, all eager algorithms need to maintain additional storage space for holding extra data structures. Specifically, SHJ needs to maintain two hash tables for both input streams and PMJ needs to keep sorted sublists during processing. Comparing PMJ^{JB} and PMJ^{JM}, we can see that JM leads to higher memory consumption than JB. This matches findings from previous work [27] as JB only partially replicates the partition of S to a group of threads. However, this difference is not obvious between SHJ^{JB} and SHJ^{JM} as the size of the maintained hash table of one stream (e.g., R) already exceeds L3 cache size significantly. JB consumes even more memory initially because of the additional status maintenance, while JM consumes more memory later because of the growing size of the hash table of S. Second, the lazy algorithms have similar memory consumption. Sort-based ones (i.e., MPass and MWay) consume slightly more memory as they require additional space for intermediate data during shuffling and merging phases. NPJ consumes more memory than PRJ as the size of the shared hash table significantly exceeds the last level cache. This result reaffirms the previous analysis of the drawback of NPJ [5].

Table 5 shows the hardware performance counters per input tuple, which reaffirms our previous findings. For example, NPJ shows high cache misses because of the size of the shared hash table is larger than the cache size, almost every access to the hash table results in a cache miss. Table 6 further shows the hardware resource utilization among different algorithms. Except for NPJ, all lazy algorithms generally show relatively lower CPU utilization because they need to wait until all data arrive, resulting in a significant idle time. The NPJ consumes a large portion of CPU cycles in dealing with cache misses and hence show high CPU utilization. Due to their aggressive process nature, the eager algorithms generally consume more resources in terms of both CPU and memory bandwidth. This indicates that increasing memory bandwidth and CPU power can further improve their performance. We will confirm this in the subsequent multicore scalability study.



(a) Execution Time Breakdown

(b) Memory Consumption

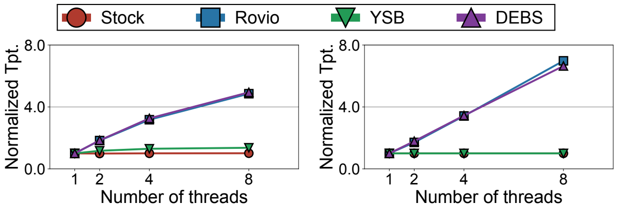
Figure 19: Micro-architectural Analysis on Rovio.

Table 5: Counters per input tuple (Rovio)

	NPJ	PRJ	MWay	MPass	SHJ^{JM}	SHJ^{JB}	PMJ^{JM}	PMJ^{JB}
TLBD Misses	0.686	0.027	0.020	0.028	1.312	2.144	0.764	0.172
TLBI Misses	0.740	0.037	0.009	0.010	2.431	3.361	0.450	0.436
L1I Misses	15.976	1.590	0.212	0.222	24.904	26.445	3.636	1.554
L1D Misses	17529.971	541.168	2442.201	2438.575	17238.905	14540.641	1151.674	685.827
L2 Misses	6566.433	10.145	0.238	0.310	5080.151	4445.801	63.268	27.975
L3 Misses	3028.627	0.038	0.020	0.052	3854.870	3691.542	0.249	0.446
Branch Misp.	1.571	0.515	1.008	0.873	5.257	3.994	1.000	1.179
Instr. Exec.	8876.563	8793.415	3202.879	3205.110	17763.657	14809.760	17723.712	14649.200

Table 6: Resource utilization (Rovio)

	NPJ	PRJ	MWay	MPass	SHJ^{JM}	SHJ^{JB}	PMJ^{JM}	PMJ^{JB}
Mem. BW.(%)	19.093	0.144	0.302	0.480	23.978	20.306	1.444	0.473
CPU. Util.(%)	98.456	24.606	55.346	59.163	99.045	80.431	97.287	90.752



(a) MPass (Lazy)

(b) SHJ^{JM} (Eager)

Figure 20: Impact of multicores.

Multicore Scalability. We now show the multicore scalability in terms of throughput. Our experiments show that MPass has slightly better scalability compared to MWay. Hence, we take MPass and SHJ^{JM} as examples to represent the lazy and eager approach here. The results are shown in Figure 20. As expected, both algorithms are not affected by varying hardware resources when handling **Stock** and **YSB** as the system is underutilized. For **Rovio** and **DEBS**, SHJ^{JM} scales slightly better. We observe that this is due to its eager nature, which aggressively utilizes available hardware resources whenever possible (i.e., as soon as any input tuples arrive). In contrast, MPass can only start to utilize all resources when all input tuples are ready.

Impact of SIMD. AVX utilizes CPU registers to perform a Single Instruction on Multiple pieces of Data (SIMD) to accelerate data processing. We now examine the effect of AVX instruction sets used in MWay, MPass, PMJ^{JM} and PMJ^{JB}. Note that, our benchmark currently only utilize AVX-256 (inherited from the existing codebase from previous work [5]), further supporting AVX-512 is expected

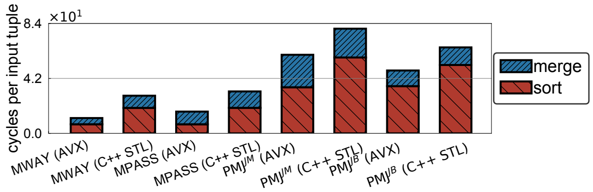


Figure 21: Impact of SIMD.

to bring more performance improvement but left as future work. We use **Micro** (static datasets) in this study. Figure 21 clearly demonstrates the benefit of utilizing SIMD in reducing the cost of sorting. It also reduces the cost of merging slightly in MPass and MWay. The overall improvement varies from 1.2x to 2.5x, which matches the observations from the existing study [5]. The improvement from SIMD on $\mathtt{PMJ}^{\mathtt{JM}}$ and $\mathtt{PMJ}^{\mathtt{JB}}$ is less significant (1.2x) as it is mainly memory bounded for handling this workload. Thus, to further improve the performance of PMJ, one has to focus more on improving its data access pattern.

6 RELATED WORK

In this section, we review related work and reveal the limitations that motivate this work.

Stream Joins. For specific stream join algorithms, prior works such as [32, 38, 44] have explored different ways to utilize multi-core or many-core processors, but their primary concern is to efficiently and incrementally process subsequent sliding windows. For example, the handshake join [44] and SplitJoin [32] are proposed to better utilize modern multicore architectures with a dataflow model, where each join core has to constantly update its internal state. The recently proposed IBWJ [40] accelerates *inter-window join* by utilizing a shared index structure, which brings performance gains during tuple matching but brings even higher state maintenance costs as the shared index structure needs to be constantly updated during processing. In contrast, we focus on *intra-window join* (*IaWJ*), assuming there is *no* subsequent windows at all. The inter- and intra-window joins have a very different design goal and are hardly comparable. To validate our analysis, we have implemented and evaluated the handshake join [44] and observed that it leads to orders of magnitude lower throughput than any of the eight algorithms that we have evaluated. This is due to the additional overhead for maintaining window updates. Early work of *IaWJ* historically focuses on its single-thread execution efficiency [13, 49] and is no longer suitable on modern multicore architectures. Recent studies [14, 27] have proposed parallel *IaWJ* algorithms by wisely distributing input tuples of the same window among parallel threads without considering window updates. Different from them, our study is the first attempt to systematically evaluate different approaches (including both relational join algorithms and specific stream join algorithms) to parallelize *IaWJ* on modern multicore processors.

Relational Joins. To optimize relational joins, past works have explored the usage of SIMD instructions [25], cache- and TLB-aware data replication [5, 6], memory-efficient hash table [7] and NUMA-aware data placement strategies [3]. Extensive recent efforts have also been devoted to accelerating join processing by better utilizing specialized hardware architectures such as GPU [28, 42], FPGA [10, 17] and high-bandwidth memory [34]. The common goal is to find a scheme that minimizes the overall join processing time by wisely managing the balance between computation and communication/partitioning cost among different architectures. All those join algorithms are covered by our concerned algorithm design aspects including *eager/lazy*, *sort/hash*, and various *partitioning schemes*, but with novel implementations such as specialized data partitioning schemes for novel hardware architectures [28, 34, 42] and adaptive partitioning models [24]. All those works are highly valuable, but none of them has comprehensively evaluated different approaches to parallelize *IaWJ* on modern multicores. Nevertheless, our open-sourced benchmark enables the community to evaluate more workloads, join algorithms, and hardware platforms in future. Schuh et al. [39] recently studied 13 relational join implementation variants, where PRJ and MWay are treated as the state-of-the-art general purpose hash- or sort-based relational join algorithms, respectively. Both PRJ and MWay have been included into our benchmark as the representative lazy algorithms. Schuh et al. [39] have further proposed two PRJ variants (called PRLiS and PRAiS, respectively) and one novel algorithm called Chunked Parallel Radix partition (CPRL). These algorithms are especially optimized towards NUMA, and have demonstrated excellent performance on multi-socket servers. In this paper, we focus on single socket multicore processors and leave the evaluation of NUMA optimization such as CPRL to future work.

7 CONCLUSION

In this paper, we present results from an extensive experimental analysis of existing join algorithms across both relational join algorithms (i.e., the lazy approach) and specific stream join algorithms (i.e., the eager approach) for parallelizing the *intra-window join* (*IaWJ*) operation on modern multicore processors. With a comprehensive set of workloads, our experimental results highlight many important insights that have not been discussed in previous studies. In summary, we find that none of the existing join algorithms performs best in all cases for parallelizing the *IaWJ* operation. Workload characteristics, performance metrics, and hardware architectures should be considered when applying the right implementation. Based on our analysis, we have also proposed a decision tree that can be used to guide the selection of an appropriate algorithm.

This work highlights a number of directions for future work: (i) there is a need for developing an adaptive *IaWJ* algorithm that considers all the factors including workload, metrics and hardware, (ii) it is important to further extend this study to include more hardware architectures such as NUMA, HBM, GPUs, and FPGAs, and (iii) our successful attempt of using relational join algorithms to accelerate joining over data streams indicates the great potential of joint efforts from different communities to better support modern data intensive stream applications. This is a general problem with lots of potential for which we lay the foundation with this paper.

ACKNOWLEDGMENTS

This work was funded by the DFG Priority Program (MA4662-5), the German Ministry for Education and Research as BIFOLD (ref. 01IS18025A and ref. 01IS18037A) , and a SUTD Start-up Research Grant (SRT3IS21164).

REFERENCES

[1] 2018. *Interval Join in Apache Flink , https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/joining.html*. Last Accessed: 2020-09-17.

[2] 2018. *Shanghai Stock Exchange, http://english.sse.com.cn/*.

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

[4] Jieliang Ang, Tianyuan Fu, Johns Paul, Shuhao Zhang, Bingsheng He, Teddy Sison David Wenceslao, and Sienyi Tan. 2019. TraV: an Interactive Trajectory Exploration System for Massive Data Sets. In *Proc. BIGMM.* 309–313.

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE.* 362–373.

[7] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 353–364.

[8] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proc. SIGMOD.* 37–48.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 36, 4 (2015).

[10] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA Useful for Hash Joins?. In *CIDR.*

[11] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1313–1324.

[12] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPSW.* IEEE, 1789–1792.

[13] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. 2002. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *Proc. VLDB.* 299–310.

[14] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 441–452.

[15] Vincenzo Gulisano, Zbigniew Jerzak, Spyros Voulgaris, and Holger Ziekow. 2016. The DEBS 2016 grand challenge. In *Proc. DEBS.* ACM, 289–292.

[16] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. *SIGMOD Rec.* 28, 2 (June 1999), 287–298.

[17] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR.*

[18] Intel. 1997. *Read Time-Stamp Counter, https://c9x.me/x86/html/file_module_x86_id_278.html*. Last Accessed: 2020-06-29.

[19] Intel. 2016. *Intel 64 and IA-32 Architectures optimization Reference Manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf*.

[20] Intel. 2017. *Intel Performance Counter Monitor, 2017, https://software.intel.com/en-us/articles/intel-performance-counter-monitor*. Last Accessed: 2020-06-29.

[21] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, Anirban Banerjee, Benjamin Heintz, Shridar Iyer, and Anshul Jaiswal. 2018. Providing Streaming Joins as a Service at Facebook. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1809–1821.

[22] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *ICDE.* IEEE, 1507–1518.

[23] Parag Kesar and Ben Liu. 2019. *Real-time-experiment-analytics-at-pinterest-using-apache-flink, https://medium.com/pinterest-engineering/real-time-experiment-analytics-at-pinterest-using-apache-flink-841c8df98dc2*.

[24] Omar Khattab, Mohammad Hammoud, and Omar Shekfeh. 2018. PolyHJ: A Polymorphic Main-Memory Hash Join Paradigm for Multi-Core Machines. In *Proc. CIKM.* 1323–1332.

[25] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.

[26] Ramon Lawrence. 2005. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In *Proc. VLDB.* 841–852.

[27] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proc. SIGMOD.* 841–852.

[28] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proc. SIGMOD.* 1633–1649.

[29] Tiziano Matteis and Gabriele Mencagli. 2017. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *Int. J. Parallel Program.* 45, 2 (April 2017), 382–401.

[30] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. 2004. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *ICDE.* 251–.

[31] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting Networks on FPGAs. *The VLDB Journal* 21, 1 (Feb. 2012), 1–23.

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

[33] Pinterest. 2020. *Pinterest, https://www.pinterest.com/*. Last Accessed: 2020-11-23.

[34] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. 2019. Joins on high-bandwidth memory: a new level in the memory hierarchy. *The VLDB Journal* (2019), 1–21.

[35] Linux Project. 2020. *Performance analysis tools for Linux, https://man7.org/linux/man-pages/man1/perf.1.html*. Last Accessed: 2020-11-24.

[36] Yuan Qiu, Serafeim Papadias, and Ke Yi. 2019. Streaming HyperCube: A Massively Parallel Stream Join Algorithm. In *EDBT.* 642–645.

[37] Rovio. 2019. *Creator of the Angry Birds game, http://www.rovio.com/*. Last Accessed: 2020-06-29.

[38] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency Handshake Join. *Proc. VLDB Endow.* 7, 9 (May 2014), 709–720.

[39] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proc. SIGMOD.* 1961–1976.

[40] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-Based Stream Join on a Multicore CPU. In *Proc. SIGMOD.* 2523–2537.

[41] shuhao Zhang. 2020. *Our Benchmark Suite, https://github.com/ShuhaoZhangTony/AllianceDB*. Last Accessed: 2020-11-10.

[42] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE.* 698–709.

[43] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. 2005. RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization. In *Proc. SIGMOD.* 371–382.

[44] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. In *Proc. SIGMOD.* 625–636.

[45] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proc. SIGMOD.* 147–156.

[46] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT.* 97–108.

[47] Tolga Urhan and Michael J Franklin. 2000. Xjoin: A reactively-scheduled pipelined join operatorỳ. *Bulletin of the Technical Committee on* (2000), 27.

[48] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proc. SOSP.* 374–389.

[49] A. N. Wilschut and P. M. G. Apers. 1991. Dataflow query execution in a parallel main-memory environment. In *Proc. ICPADS.* 68–77.

[50] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *ISPASS.* 35–44.

[51] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR.*

[52] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 516–530.

[53] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *ICDE.* 659–670.

[54] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2020. Hardware-Conscious Stream Processing: A Survey. *SIGMOD Rec.* 48, 4 (Feb. 2020), 18–29.