SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision

Mohammadreza Najafi, *Technische Universität München;* Mohammad Sadoghi, *IBM T. J. Watson Research Center;* Hans-Arno Jacobsen, *Middleware Systems Research Group*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/najafi

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision

Mohammadreza Najafi[†], *Mohammad Sadoghi*[‡], *Hans-Arno Jacobsen*[§]
[†]*Technical University Munich*
[‡]*IBM T.J. Watson Research Center*
[§]*Middleware Systems Research Group*

Abstract

There is a rising interest in accelerating stream processing through modern parallel hardware, yet it remains a challenge as how to exploit the available resources to achieve higher throughput without sacrificing latency due to the increased length of processing pipeline and communication path and the need for central coordination. To achieve these objectives, we introduce a novel top-down data flow model for stream join processing (arguably, one of the most resource-intensive operators in stream processing), called SplitJoin, that operates by splitting the join operation into independent storing and processing steps that gracefully scale with respect to the number of cores. Furthermore, SplitJoin eliminates the need for global coordination while preserving the order of input streams by re-thinking how streams are channeled into distributed join computation cores and maintaining the order of output streams by proposing a novel distributed punctuation technique. Throughout our experimental analysis, SplitJoin offered up to 60% improvement in throughput while reducing latency by up to 3.3X compared to state-of-the-art solutions.

1 Introduction

Scalable stream processing is an integral part of a growing number of data management applications such as real-time data analytics [1], algorithmic trading [2], intrusion detection [3], and targeted advertising [4]. These latency-sensitive and throughput-intensive applications have motivated database research to seek new avenues for accelerating data management operations in general and stream processing in particular. These new approaches have adopted heterogeneous architectures (*e.g.*, GPUs and Cell processors) [5, 6, 7, 8], multi-core architectures [9, 10, 11, 12, 13], and Field Programmable Gate Arrays (FPGAs) [2, 14, 15, 16, 17, 18, 19, 20] for stream processing acceleration.

Besides leveraging hardware acceleration, coping with the high-velocity of unbounded incoming streams has forced the stream operation model to shift away from the traditional "store and process" model that has been prevalent in database systems for decades. However, the mindset of sequential stream join processing (or constructing lengthy processing pipelines) and, essentially, thinking of a stream as a sliding window (or a long chain of sequentially incoming tuples to resemble database relations) has continued to shape the way stream processing is carried out today, even on low-latency and high-throughput stream processing platforms.

Stream Join Challenges: To mitigate the challenges imposed by unbounded streams, with respect to both processing and space constraints, data streams are conceptually seen as bounded sliding windows of tuples (*i.e.*, simulating a relation). Sliding windows are defined as a function of time or as a fixed number of tuples. Once the sliding window abstraction is set (*i.e.*, tuples are admitted for processing), the stream join semantics over the windows are identical to the traditional join semantics in relational database systems.

Although the sliding window provides a robust abstraction to deal with the unboundedness of data streams [6, 9, 10, 11, 12, 13, 16, 19], it remains a challenge to improve parallelism within stream join processing, especially, when leveraging many-core systems. For example, a single sliding window could conceptually be divided into many smaller sub-windows, where each sub-window could be assigned to a different join core.[1] However, distributing a single logical stream into many independent cores introduces a new coordination challenge: to guarantee that each incoming tuple in one stream is compared exactly once with all tuples in the other stream.

The coordination challenge is addressed by handshake join [9] that transforms the stream join into a bi-directional data flow problem: tuples flow from left-to-right (for S stream) and from right-to-left (for R stream)[2] and pass through each join core. The bi-directional data flow ensures that every tuple is compared exactly once by design (shown in Figure 1). This new data flow model offers greater

[1] A join core is an abstraction that could apply to a processor's core, a compute node in a cluster or a custom-hardware core on FPGAs (*e.g.*, [9, 19, 16, 10].)

[2] The join operator is performed on two streams which we refer to as S and R streams.

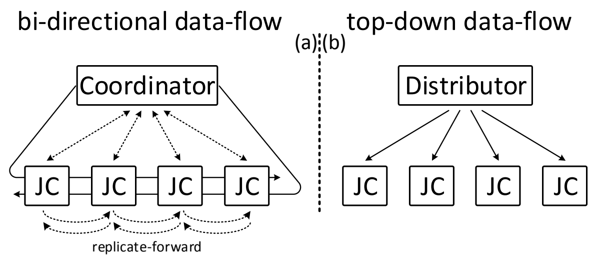

Figure 1: Stream join data flow models (*JC* stands for join core): (a) bi-directional and (b) top-down.

processing throughput by increasing parallelism, yet suffers from latency increase since the processing of a single incoming tuple requires a sequential flow through the entire processing pipeline. To improve latency, yet another central coordination is introduced to fast-forward tuples through the linear chain of join cores in the low-latency handshake join [10] (the coordination module is depicted in Figure 1). For each stream, in order to reduce latency, once a tuple reaches a join core, the tuple is replicated and forwarded to the next core before the join computation is carried out [10].

Generally speaking, any central coordination prohibitively limits the scalability of processing as the degree of parallelism is increased (*cf.* Amdahl's Law); and, central coordination is required in [6, 9, 10, 19]. For example, the coordinator must explicitly send expiration messages to each join core as new tuples enter and old tuples leave each sub-window assigned to a join core. Furthermore, the flow of new and expired tuples in and out of cores is further complicated if tuples are additionally replicated and fast-forwarded [6, 10]. Consequently, the neighboring cores must explicitly communicate (in addition to communicating with the global coordinator) and, in fact, all tuples from both streams actually pass through this communication channel [9, 10]. Moreover, an explicit knowledge of the underlying hardware is required (that may not even be available in virtual machine settings), and one must rely on a complex optimal assignment of the join cores to physical cores to reduce the NUMA-effect by reducing the size of communication paths between neighboring cores [9, 10].

Problem Statement: In this paper, we tackle two main shortcomings of existing stream join processing architectures: the sequential operation model (*i.e.*, "store" and "process") and the linear data flow model (*i.e.*, "left-to-right" and "right-to-left" flows). We propose SplitJoin, the first step in re-thinking the stream join operation model, which is built on the implicit assumption that storage of newly incoming data, whether stored in a relation or a memory buffer, must always precede processing. Instead, we abstract the computation steps as two independent and concurrent steps, namely, (i) "storage" and (ii) "processing".[3] This new splitting abstraction of join cores enables unprecedented scalability by allowing the system to distribute the execution across many independent storage cores[4] and processing cores. Second, we change the way tuples enter and leave the sliding windows, namely, by dropping the need to have separate left and right data flows (*i.e.*, bi-directional flow). SplitJoin introduces a novel top-down data flow (*i.e.*, a single flow), where incoming tuples (from both streams) are simply arriving via the same path downstream (preserving input stream order), while the join results are further pushed and merged downstream using a novel relaxed adjustable punctuation (RAP) technique (preserving the output stream order). Unlike recent advances in stream join processing [6, 9, 19, 10], SplitJoin does not rely on central coordination for propagating and ordering the input/output streams.

SplitJoin's top-down data flow trivially satisfies the ordering of incoming tuples and eliminates the in-flight race condition between the left and right streams as tuples travel from one core to the next. Unlike existing approaches [9, 10], the top-down flow also eliminates the need for communication between the join cores. In SplitJoin, the top-down flow is realized using a distribution tree for routing incoming tuples into their corresponding sub-window that addresses the scaling issues of adding new join cores. The adopted distribution mechanism nicely fits into the coordination-free protocol of SplitJoin for distributing new tuples to both storage and processing cores. For example, all join cores receive the newly incoming tuples (achieving the desired expedited delivery, without the linear forwarding used in [10]), while only one storage core stores the new tuple. Both the storage and eviction of tuples to and from cores are done in a round-robin fashion; thus, naturally, in the same order that cores store a new tuple, they evict their oldest tuple (again, without any explicit coordination). This can be generalized to batches of tuples instead of a single tuple as well.

SplitJoin has provably lower runtime complexity as compared to state-of-the-art parallel distributed join algorithms [9, 10]. SplitJoin exhibits an overall system latency of $O(\log_b k)$, where k is the number of join cores and b is the branching factor of the distribution tree. In contrast, the state-of-the-art handshake join has $O(k)$, while the original version resulted in an $O(n)$ latency, where n is the number of tuples a window can hold ($k \ll n$) [9, 10].

SplitJoin's coordination-free distribution also lends itself to a simpler resiliency against failures; for example, core failures do not halt or disrupt the entire join computation and affect only the failed nodes (the loss is limited to only failed nodes). In contrast, in a linear left-to-right data flow, if any cores fails, then, on average, half of the cores may not receive any data.

SplitJoin is comprised of the following core components: a distribution network, a set of independent join cores, and a

[3]In relational databases, tuples are first stored in relations prior to being processed (*e.g.*, performing a join) while in a stream join, the incoming tuples are first processed and subsequently stored in sliding windows [21].

[4]A storage core is an abstraction for an in-memory sliding window, tightly coupled with a join core.

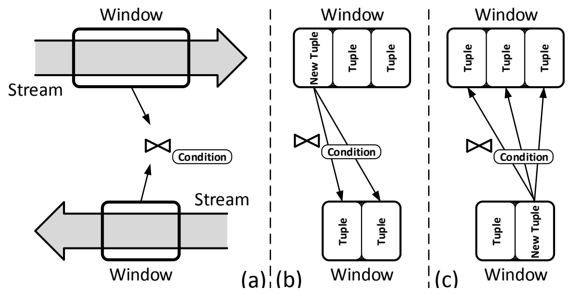

Figure 2: Sliding window concept in stream join.

result gathering network. The distribution network broadcasts incoming tuples to the set of join cores in a scalable way. The actual join computation is carried out by each join core independently, and subsequently, the joined tuples are pushed down to the result gathering network. The result gathering network is further responsible to ensure the correct ordering of the joined tuples by using the punctuation marks produced by the join cores.

In this paper, we make the following contributions:

(1) we propose SplitJoin, a novel scalable stream join architecture that is highly parallelizable and removes inter-core communications and dependencies,

(2) we introduce a new splitting abstraction in SplitJoin to "process" and "store" incoming data streams concurrently and independently,

(3) we propose a top-down data flow model to achieve a coordination-free protocol for distributing and parallelizing stream join processing,

(4) we develop a distribution tree with logarithmic access-latency for routing of incoming data to storage and processing cores, while preserving the ordering of incoming tuples,

(5) we design a coordination-free protocol that does not rely on global knowledge to produce ordered join output streams by proposing a relaxed adjustable punctuation (RAP) technique with tunable precision, and

(6) we conduct an extensive analytical and experimental study of SplitJoin as compared to existing state-of-the-art solutions.

2 Preliminaries

The relational join (theta join) between two non-stream relations R and S, defined as $R \bowtie_\theta S$, produces the set of all resulting pairs (r,s), which satisfy the join condition $\theta(r,s)$ and $r \in R$, $s \in S$. Extending this definition to stream join implies the same join processing semantics with the exception that streams, unlike relations, are unbounded. To mitigate the challenge of unbounded streams, with respect to both processing and storage limitations, streams are conceptually seen as bounded sliding windows of tuples, as shown in Figure 2. The size of these windows are defined as a function of time or number of tuples, referred to as time-based or count-based windows, respectively.

Figure 3 shows the traditional architecture of a join

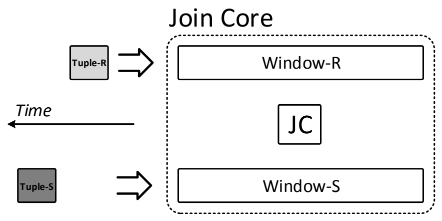

Figure 3: Traditional stream join architecture.

operator that receives Tuple-R and Tuple-S from streams R and S, respectively. *JC* stands for join core, which performs the join operation. To process the tuples shown in the figure, Tuple-R is inserted into *Window-R*, then it is evaluated against all existing tuples in *Window-S* and the join results are returned. Similarly, Tuple-S is inserted into *Window-S* and the same join procedure is applied.

3 Related Work

Work related to our approach can be broadly classified into stream join algorithms [6, 9, 10, 16, 21, 22], or more generally speaking, stream processing in software [23, 24, 25], approaches to performance-optimize stream processing through emerging hardware mechanisms [26], in particular, through FPGA-based acceleration [15, 17], but also, through GPUs and processor-based I/O processing innovations [8]. The survey [27] covers other related work and topics including concepts such as ordering in stream join. SplitJoin can be incorporated in any of the existing streaming engines (*i.e.*, [28, 29, 30]).

Stream Join Algorithms — An early stream join was formalized by Kang's three-step procedure [21]. Subsequently, Gedik *et al.* [6] introduced the parallel CellJoin, designed for a heterogeneous architecture, aiming to substantially improve stream join processing performance. However, CellJoin requires a re-partitioning task for each newly incoming tuple, which limits its scalability [6]. The problem of distributed stream join processing has also been studied with respect to elasticity and reduction of memory footprint, applicable to cloud computing [22].

Teubner *et al.* introduced a bi-direction data flow-oriented stream join processing approach, called the handshake join [9]. To reduce delay in the linear chaining, Teubner *et al.* [10] introduced a low-latency handshake join that uses a fast forwarding mechanism to expedite tuple delivery to all sub-windows by replicating every tuple k times, where the stream is split over k join cores. This mechanism is illustrated in Figure 10. Furthermore, the bi-directional flow complicates the logic for serializing the two pipes connecting consecutive join cores that is necessary in order to avoid race conditions due to concurrent in-flight tuples (*i.e.*, tuples traveling between neighboring processing cores).

Stream Processing Acceleration — Stream processing has received much attention over the past few years. Many viable research prototypes and products have been

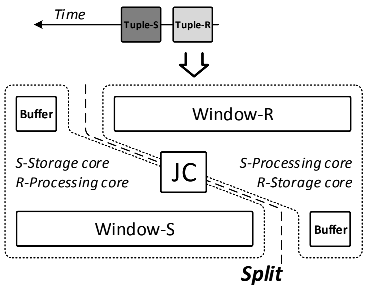

Figure 4: SplitJoin concept.

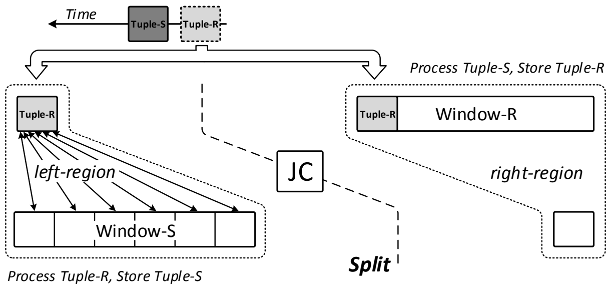

Figure 5: SplitJoin storing and processing steps.

developed, such as NiagaraCQ [24], TelegraphCQ [23], and Borealis [25], to just name a few. Most existing systems are fully software-based and support a rich query language, but stream join acceleration has not been the main focus of these approaches.

Since the inception of stream processing, the development of optimizations both at the query-level and at the engine-level have been widely explored. For example, co-processor-based solutions utilizing GPUs [8, 6] and more recently hardware-based solutions employing FPGAs have received attention [14, 15, 17, 18, 31]. For example, Tumeo *et al.* demonstrated how to use GPUs to accelerate regular expression-based stream processing language constructs [8]. The challenge in utilizing GPUs lies in transforming a given algorithm to use the highly parallel GPU architecture that has primarily been designed to perform high-throughput matrix computations and not, foremost, low latency processing.

Past work showed that FPGAs are a viable option for accelerating certain data management tasks in general and stream processing in particular [14, 15, 17, 18, 31, 32, 33]. For example, Hagiescu *et al.* [15] identify compute-intensive nodes in the query plan of a streaming computation. To increase performance in the hardware design that realizes the streaming computation, these nodes are replicated, which, due to the stateless nature of the query language considered, poses few issues. A main difference from our work is the restriction to stateless operations and the lack of a capability to flexibly update the streaming computation. Similarly, Mueller *et al.* [17] present Glacier, a component library and compiler, that compiles streaming queries into logic circuits on an operator-level basis. Both approaches are characterized by the goal of hardware-aware acceleration of streams, yet our solution is also applicable to non-FPGA parallel hardware.

4 SplitJoin

In this section, we describe SplitJoin and highlight two of its key properties, namely, the top-down data flow and the splitting of the join computation into independent storage and processing steps. Together, these properties remove any need for coordination and dependencies among join cores, which enables a high-degree of parallelism for SplitJoin without sacrificing latency.

4.1 SplitJoin Overview

SplitJoin diverts from the bi-directional data flow-oriented processing of existing approaches [9, 10]. As illustrated in Figure 1, SplitJoin introduces a single top-down data flow that fundamentally changes the overall tuple processing architecture. First, the join cores are no longer chained linearly (*i.e.*, avoiding linear latency overhead). In fact, they are now completely independent (*i.e.*, also avoiding inter-core communication overhead). Second, both streams travel through a single path entering each join core; thus, eliminating all complexity due to potential race conditions caused by in-flight tuples and complexity due to ensuring the correct tuple-arrival order, namely, the FIFO property is trivially satisfied by using a single (logical) path. Third, the communication path can be fully utilized to sustain the maximum throughput and each tuple no longer needs to pass every join core.

Another important aspect of SplitJoin is the simplification and decomposition of join processing itself. SplitJoin splits the dominant join abstraction that enforces the "storing" and "processing" steps to be coupled and done in a serial order. SplitJoin views these steps as two independent steps, namely, (i) "storing" and (ii) "processing". In fact, SplitJoin goes one step further and shows that not only these steps could be done in parallel, they can also be distributed to independent join cores. Therefore, unlike traditional parallel join processing that divides a single window into a set of sub-windows, where each is assigned to a core, SplitJoin introduces separate *storage* and *processing* cores that operate independently of each other as shown in Figure 4. The storage core is responsible for storing new tuples, while the processing core is responsible for the actual join operation of a new tuple in one stream with the existing tuples in the other stream.

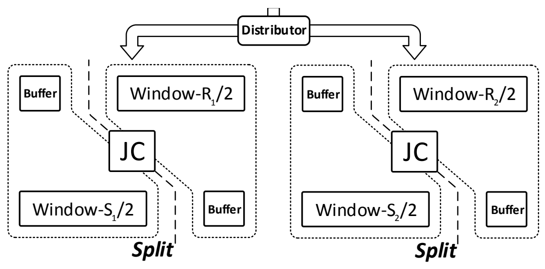

Figure 6: SplitJoin parallel architecture.

The splitting line in Figure 4 conceptually divides our join processing architecture into two separate parts, in which a *region* represents a stream's window and the associated buffer. We use the term *right-region* when referring to *Window-R* and *left-region* for *Window-S*. For each incoming tuple, a region either does processing or storing.

The split mechanism is illustrated in Figure 5, where the incoming tuples are fed to SplitJoin one after another. In the first step, `Tuple-R` is inserted into both regions. The right-region is responsible for storing `Tuple-R` in its sliding window, while the left-region is responsible for the processing of the replicated copy of `Tuple-R` (*i.e.*, the join comparison). The temporary tuple replication eliminates all inter-region communication among storage and processing cores. The replicated tuples are simply discarded once the processing is completed.

4.2 SplitJoin Parallelism

In SplitJoin, we parallelize the stream join computation by dividing each sliding window into a set of disjoint sub-windows. Each sub-window is assigned independently to a join core as shown in Figure 6 (*i.e.*, acting as a local buffer for each core). Each join core (*JC*) consists of a *left-* and a *right-region*. The division of the sliding window among join cores is accompanied by a *Distributor* unit to transmit incoming tuples to the join cores.

In the parallelized version of SplitJoin, all join cores receive the new incoming tuple. In each join core, depending on the tuple origin *i.e.*, whether *R* or *S* stream, the processing and storage steps are orchestrated. For example, if the incoming tuple belongs to the *R* stream, `Tuple-R`, then all processing cores dedicated to the left-region compare `Tuple-R` against all the tuples in the *S* stream sub-windows. Simultaneously, `Tuple-R` is also stored in the storage core of exactly one right-region. The assignment of `Tuple-R` follows an arbitration of the tuple to a storage core based on a round-robin selection. In other words, each region, based on its position number [5] and the number of seen tuples, independently determines its turn to store an incoming tuple. The proposed assignment model eliminates the need for a central coordinator for tuple assignment, which is a key contributor for achieving scalability in SplitJoin architecture. Notably, transmitting an incoming tuple to each join core translates into writing a tuple to the join core's local buffer (independent of any other join cores) that resembles a simple queue with a single producer and a single consumer, in which the producer is the *Distributor* and the consumer is join core itself.

4.3 Scalable Distribution Tree

The decoupling of storage and processing in SplitJoin simplifies parallelization by distributing sub-windows among many independent join cores. To fully leverage potential parallelism, we also need an efficient tuple distribution and routing mechanism.

In SplitJoin, to distribute the stream's transmission load in a balanced and scalable manner, we use a *k-ary* tree as the distribution network. As the network grows in size, the *Distributor* is replicated and its replicas are placed in the tree's inner nodes to achieve the desired scalability. As the number of SplitJoin join cores increases, we increase the fanout of each *Distributor* before increasing the depth of the distribution tree.

By applying replication recursively, we scale the distribution network as well as the number of join cores for SplitJoin. The resulting system, including the input data distribution network, SplitJoin's join cores, and the output data gathering network (similar in structure to the input network), is shown in Figure 11, where the horizontal bars illustrate the input distribution and output gathering networks.

The distribution network is the same for both count-based and time-based sliding window joins. However, in the time-based version each tuple carries an extra field for its timestamp. This field is to keep track of the lifespan of each tuple to realize the time-based sliding window semantic.

4.4 Expiration & Replacement Policies

Tuple expiration is a crucial step to ensure the correctness of the stream join semantic. In the count-based sliding window, the number of tuples in each window is specified explicitly while in the time-based sliding window a lifespan l (*e.g.*, $l = 10$ minutes) defines when a tuple must be expired.

SplitJoin supports both passive and active expiration techniques. The passive approach is primarily intended for the count-based sliding window, in which the incoming tuples simply overwrite the oldest tuples in the window. The expiration is done implicitly and mimics the functionality of a FIFO buffer. Once a window is full, the stored tuples are expired in order of their arrival. In the active expiration, geared towards the time-based sliding window, each join core locally manages the expiration of tuples from its sub-window. The expiration task for each sub-window is postponed until a tuple from the opposing stream with timestamp of t is received for processing. Then, in the region responsible for processing, just prior to the join computation, for each tuple with a timestamp t_i, if $(t - t_i) > l$, then the tuple is expired. Basically, tuples are expired when they fall off the user-defined lifespan (l) of the time-based window size.

Note the expiration is a local operation within a region and does not involve global coordination because tuples arrive with monotonically increasing timestamps and order is preserved when they are added and stored in a sub-window. The expiration task starts from the end (with the oldest tuple) of each sub-window and ends when a tuple younger than the user-defined lifespan is found. In other words, instead of sending explicit expiry messages with a timestamp, we rely on the timestamp of tuples in the input streams that must

[5] Position number refers to the logical location of a join core among other join cores.

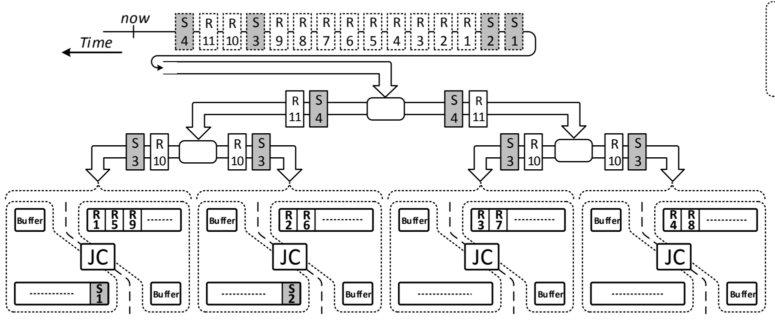
Figure 7: SplitJoin data distribution and processing example.

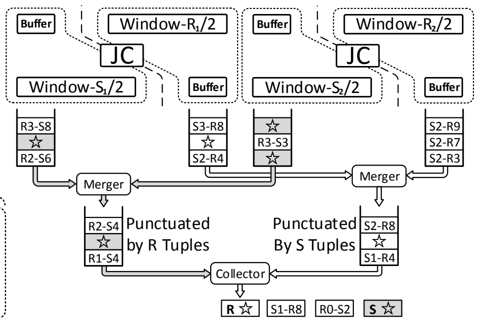

Figure 8: Punctuations in result gathering.

Algorithm 1: SplitJoin distribution network.

```
SplitJoin() begin
    while still a tuple to consume do
        broadcast tuple t to all Join Cores begin
            forall join cores do
                Join_Core(t, source);
```

be routed to all nodes anyway. Therefore, the expiration messages are implicitly piggybacked on the incoming tuples as a way to broadcast the synchronized time without the need for global coordination.

In Figure 7, we illustrate how tuples are stored and processed in SplitJoin join cores. Assuming that we have a sequence of tuples as shown in the upper part of the figure, each tuple is transferred by the distribution tree to all join cores. Each `Tuple-R` is stored in exactly one *right-region* while processed by the *left-region* of all join cores. Likewise, for tuples from the *S* stream, they are stored in the *left-regions* and are processed by the *right-regions*.

As we can see in Figure 7, the tuples are distributed in-order. The tuples reach the storage cores through the same path (*i.e.*, the top-down flow), and the expiration procedure is preformed based on the order of incoming tuples (for both count-based and time-based sliding windows). Thus, unlike the bi-directional model used in [9, 10], neither the concurrency nor the race condition issues arise.

During processing, each region emits resulting tuples to be collected by the result gathering network (*cf.* Section 5). The processing step for each tuple in each region is completed by emitting an end notice from that region, referred to as a *star* punctuation mark. These marks serve to preserve the order of join results as we describe in detail in Section 5.

4.5 SplitJoin Algorithms

Tuple distribution in SplitJoin is specified in Algorithm 1. Upon arrival of a new tuple, regardless of its source stream, the tuple is broadcast to all join cores (Line 3).

In each join core, as presented in Algorithm 2, depending on the tuple's source (Lines 2 and 11), from *R* or *S* stream, the tuple is sent to the *right-region* for storage and to the *left-region* for processing or vice versa.

Finally, in the expiration process specified in Algorithm 4, the tuples that are too old to be considered for the join are expired from the end of the sub-window by computing their lifespan using their timestamp and the timestamp of the new tuple.

The pseudo code for the processing core (*i.e.*, the join comparison) is specified in Algorithm 3. An incoming tuple is compared with all tuples in the opposite sub-window (Lines 2-4). More importantly, this step is executed concurrently for each sub-window in every region. After processing (Lines 5-6), based on the chosen ordering precision, the *star* marker is produced and emitted. Also note that the goal of SplitJoin is to provide an efficient and coordination-free architecture for performing stream joins, and the particular choice of join algorithm is orthogonal. In this work, we adopted a simple variation of nested-loop join; however, within each core, one may choose any join algorithms such as hash- or index-based join.

5 Punctuated Result Collection

In SplitJoin, we employ a result gathering network (similar to our data distribution network) and a punctuation technique to preserve the ordering for the join result output. The full architecture of SplitJoin, that includes the distribution network, join cores (*JCs*), and the collection network, is illustrated in Figure 11.

In SplitJoin, we utilized a *2-ary* collection tree to gather and merge join results as depicted in Figure 8. The result

Algorithm 2: A join core in SplitJoin.

```
1  Join_Core(t, source) begin
2      if source = Stream R then                    // right-region
3          Expiration_Process(t, sub-window S);
4          Processing_Core(t, sub-window S);
5          if R_store_counter = node_id then
6              Storage_Core(t, sub-window R);
7          if R_store_counter = number of join cores then
8              R_store_counter ← 0;
9          else
10             R_store_counter ← R_store_counter + 1;
11     else                                         // left-region
```

Algorithm 3: Matches between **t** and sub-window **X**.

```
Processing_Core(t, sub-window X) begin
    forall t_i-tuple in sub-window X do
        compare t_i-tuple with t; if match then
            emit the matched result;
        if $i \equiv 0 \pmod{ordering\_precision}$ then
            emit punctuation star;
```

tuples of each processing core are gathered from the leaves of the collection tree. Each core has its own dedicated FIFO buffer. The collection tree employs a *Merger* unit and a FIFO buffer in each of its intermediate nodes (except in the root). Moving toward the tree's root (from top to bottom), at each node, the data in the two input buffers is merged into the buffer of that node. Merging continues up to the root, which contains the last buffer emitting the gathered join results.

5.1 Punctuation-based Ordering

SplitJoin architecture preserves the ordering of result tuples. The precision of the output order can be determined by a tunable system parameter, without significant changes in the processing architecture. To realize this flexibility in our design, we developed a *relaxed adjustable punctuation* (RAP) strategy. We define two levels of ordering guarantees for join results: the *outer* and *inner ordering*.

Definition 1 *The outer ordering of join results ensures that for any two consecutive incoming tuples, join results of the first tuple always precede the join results of the second tuple.*

Definition 2 *The inner ordering of join results ensures that for a single incoming tuple in one stream, join results are ordered in ascending order from the oldest to the most recently inserted tuple in the other stream.*

Our proposed relaxation enables us to maintain strict outer ordering while adjusting the precision of the inner ordering (essentially, not maintaining the inner ordering) in order to substantially reduce the overall cost of ordering. Furthermore, our technique supports strict outer and inner ordering as well.

In RAP, we define a simple punctuation emission rule for each core (the same simple rule applies to all cores), that is, the emission of a punctuation at the end of the processing of every newly inserted tuple (preserving the outer ordering and relaxing the inner ordering). In other words, each join core emits a punctuation after the end of processing a newly inserted tuple with all tuples in the other window. We

Algorithm 4: Expiring old tuples for time-based version.

```
Expiration_Process(t, sub-window X) begin
    $i \leftarrow$ the end of sub-window X;
    while $t_i$.timestamp - t.timestamp > Time Window Size do
        omit $t_i$ from sub-window X;
        $i \leftarrow i - 1$;
```

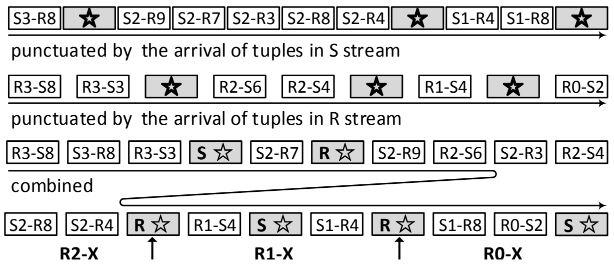

Figure 9: Punctuated resulting stream.

differentiate this punctuation from result tuples by a *star*, as shown in Figure 8.

SplitJoin cores insert both the join results and punctuation marks to collection tree leaves. The punctuation acts as a border between the join results of two consecutively inserted tuples (outer order). As join results and punctuations are pushed down the collection tree towards the root, at each node of the tree, the join results and their corresponding punctuation marker (stars) from the two buffers are merged into the FIFO buffer of their parent node. When the *Merger* in the parent node receives a star from one of its inputs, it disables that input and continues to receive resulting tuples from the other buffer until it receives a star from that buffer as well. The *Merger* merges two punctuations (stars) into one and pushes it to its FIFO buffer. This scenario repeats until the star reaches the output of the collection tree.

Since join results are pushed down in the order in which the newly inserted tuple arrives, the outer ordering for each core is trivially satisfied due to the single top-down FIFO flow of SplitJoin that starts from the root of the distribution tree (for inserting new tuples) and ends at the root of the collection tree (for merging the join results). This flow is shown in Figure 9.

The final step in the result gathering network employs a *Combiner* rather than a *Merger*. On the right side of the split are the punctuated results, ordered by the tuples from the *S* stream, while on the left side, the punctuation is based on the arrival sequences of tuples from the *R* stream. These two sets of punctuated result tuples are consumable as separate streams. However, to emit only one stream as output, we use a *Combiner* which simply fetches the resulting tuples and punctuations from their input and puts them into the output buffer. The *Combiner* keeps track of the origin of punctuations (whether from the right- or left-regions) by flagging the *stars* with *R* and *S*, as shown in Figure 9.

In Figure 9, the upper flow is the result stream from the right-regions, punctuated by the order of *S* stream tuples, while the middle one is the result stream from the left-regions, punctuated by the order of *R* stream tuples. The lower flow demonstrates the combined result stream that includes all result tuples in addition to punctuations. For example, **R1-X** is specified by two *R* punctuations and includes the result tuples which start with **R1**.

Adjusting the punctuation interval is straightforward and

Algorithm 5: Punctuation-based N-ary merger.

```
1 N-ary_Merger(t) begin
2     foreach right(or left)-region of core_{1..N} in sequence do
3         while a resulting tuple (t) is available in output buffer till the first star do
4             pop t from join core's output buffer;
5             push t to Merger's output buffer;
6     push out the end of result star;
```

only requires us to tune the punctuation emission rate in SplitJoin's cores. Each core can simply change the frequency at which a punctuation is generated. For example, each core can be tuned to produce a punctuation after joining one newly inserted tuple (strict outer ordering) or after every five tuples (relaxed outer ordering). We could also adjust the precision of inner ordering by increasing the frequency of punctuation generation. For example, to produce a strict inner ordering, each incoming tuple is compared with tuples in the opposite window (starting from the oldest to the most recently inserted one), followed by outputs for both the join result and the punctuation marker for every comparison. Therefore, if each core has a window size of w, then up to w punctuation markers (*i.e.*, *stars*) are produced for every newly inserted tuple. For a relaxed inner ordering, only one punctuation is produced after joining the incoming tuples with all the tuples in the opposite window. At the other extreme, when no ordering is required, we could simply disable the punctuation generation altogether.

5.2 Ordering Algorithm

For the result gathering network, we utilized a *k-ary* tree. Algorithm 5 specifies the pseudo-code for an *N-ary* (*e.g.*, 2-ary) *Merger* given an *N-ary* result gathering tree. The *Merger* is connected to the output FIFO buffer of N regions and collects the resulting tuples and punctuations into its own output FIFO buffer, which is subsequently fed to the next intermediate node (its parent) in the tree. This is repeated up to the root of the tree, where result tuples are punctuated by tuple arrival order from the two stream types (either R or S).

The *Merger* connects to the output buffers of the same source, either *left* or *right* regions (*cf.* Line 2 of Algorithm 5). Each *Merger* collects the results in the same order as the join cores store the new incoming tuples. For example, assuming that the first `Tuple-R` is stored in the left most join core in its *right-region*, as shown in Figure 7. The *Merger* then begins the collection of results from the comparison of `Tuple-S` with the R sub-window in the left most *right-region* as well.

In the result gathering, the *Merger* fetches tuples from the first region's buffer and stores them in its own output buffer until it reaches the first *star* in (Line 3∼Line 5). Then it repeats the same procedure for the next region's buffer until it receives a *star* from there too.

After receiving a punctuation mark from the last region, the *Merger* forwards the punctuation to its output buffer (*cf.* Line 6). Note that each *Merger* emits only one punctuation mark for every pair of punctuation (*i.e.*, one punctuation mark from each join core).

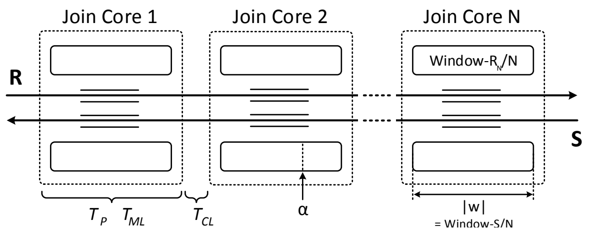

Figure 10: Low-latency handshake join overview [10].

Using a higher ordering precision increases the number of punctuation marks between result tuples of each region. For example, instead of having one punctuation mark after comparison of a tuple with the whole sub-window, we can have one punctuation after each 10 comparisons. Since tuples in the sub-window are already stored in the order of their arrival, the intermediate punctuations preserve the result ordering while gathering the results from all the join cores. For obtaining a higher precisions, *Mergers* follow the same procedure as before.

6 Runtime Complexity

In this section, we present a brief analytical model to study the runtime complexity of SplitJoin relative to related techniques [9, 10]. In the analysis, we use the following definitions.

Definition 3 *We define the processing latency (PL) as the time from when a tuple arrives at the join operator until the tuple is compared and joined with all tuples in the other window and all the matching results are produced.*

Definition 4 *We define the visiting latency (VL) as the time required for two tuples from both streams to be compared with each other.*

6.1 Low-latency Handshake Join Analysis

The processing latency for low-latency handshake join (*cf.* Figure 10) is given as follows:

$$PL = T_{CL} + ((k-1)\times(T_{CL}+T_{ML})) + (w\times T_P) + T_{Col} \quad (1)$$

where T_{CL} represents the communication time between cores and k the number of processing cores. T_{ML} accounts for the tuple monitoring time in both streams, required to prevent missing results between tuples in the fast-forwarding buffers (i.e., race conditions). Therefore, the cost of propagating a single tuple to all cores by replication and fast-forwarding is captured by $((k-1)\times(T_{CL}+T_{ML}))$. The size of each sub-window in each core is denoted by w. T_P represents the processing time to perform the join operation between each pair of tuples. To simplify the analysis, we assume that all join cores are working in parallel. Finally, T_{Col} presents the time required to collect all the matching results. In [10] the authors rely on a linear collector method for gathering the results that has the potential to break the strict neighbor-to-neighbor communication model of handshake join.

In theory, assuming a fixed-size sub-window, we can increase the number of processing cores to support larger windows. Therefore, the join's latency scales linearly in the number of processing cores, *i.e.*, as $O(k)$, — optimistically assuming that central coordination would not become a bottleneck while ignoring the effect of the result collection method.

To calculate the visiting latency, we assume that the number of in-flight tuples from the two streams that must be compared (*i.e.*, the monitoring time T_{ML}) is negligible. While this assumption renders the model less realistic (which was also implicitly assumed in [10]), it simplifies the visiting latency analysis.

Any pair of tuples from both streams meet each other in, at most, one location; let this location be α as shown in Figure 10. α could be in any core. If α happens to be on the first core, then the latency is lower, while if it is on the last core, then the latency is higher. Thus, we define the average visiting latency as follows:

$$VL_{avg} = T_{CL} + (\lfloor\frac{(k-1)}{2}\rfloor \times (T_{CL} + T_{ML})) + ((\frac{w}{2}) \times T_P) \quad (2)$$

$(\lfloor\frac{(k-1)}{2}\rfloor \times (T_{CL} + T_{ML}))$ determines the average time to reach location α (essentially, reaching the mid-point of core chain) and $(\frac{w}{2}) \times T_P$ captures the processing time for half the tuples at α. The visiting latency scales linearly with the number of processing cores $O(k)$, assuming $(T_{CL} + T_{ML})$ is constant, irrespective of the number of cores.

To simplify the analysis, we ignore the overhead of central coordination in low-latency handshake join [10]. The coordinator requires sending an explicit expiry message for every tuple [10]. On average, these messages double the communication traffic between the central coordinator and each join core, significantly affecting the performance as observed in our experimental evaluation.

6.2 SplitJoin Analysis

SplitJoin utilizes a distribution tree to deliver incoming tuples to each join core in $O(log_b k)$ time, where k is the number of join cores and b is the branching factor of the distribution tree. We define $Path_{1\cdots k}$ as a distribution route that a tuple must travel to reach the join cores $1 \cdots k$, respectively. Let $T_{C_{Path_i,Depth_j}}$ be the communication cost (duration) of transferring a tuple to the i^{th} path at depth j. We define the processing latency for SplitJoin as follows:

$$PL = \max_{i=1\cdots k}(\sum_{j=1\cdots log_b k} T_{C_{Path_i,Depth_j}} + (w \times T_{P_i})) + T_{Col} \quad (3)$$

where T_{P_i} is the processing time to perform the join operation between each pair of tuples for the i_{th} core. Assuming the communication times, $T_{C_{Path_i,Depth_j}}$, are roughly equal, then it

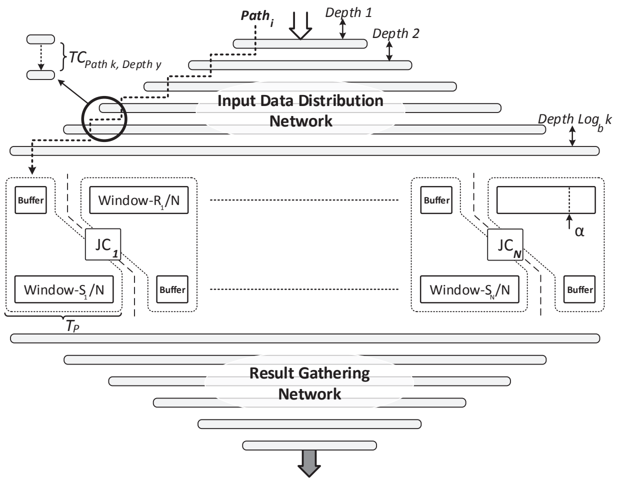

Figure 11: SplitJoin complete system.

follows that:

$$PL = \max_{i=1\cdots k}(T_{C_{Path_i}} \times log_b k) + (w \times T_{P_i})) + T_{Col} \quad (4)$$

If we further assume homogeneous join cores and homogeneous distribution routes within the tree and also decompose T_{Col} into smaller units of work, then it follows that:

$$PL = (T_{CL} \times log_b k) + (w \times T_P) + (T_{CL} \times log_c k) \quad (5)$$

where $\log_c k$ defines the depth of the result gathering tree with the branching factor of c (from root to leaves). Assuming a fixed-size sub-window, as we increase the number of join cores, latency increases logarithmically, $O(\log_b k)$ (assuming $b < c$), for SplitJoin as opposed to the linear increase ($O(k)$) observed in [10].

Supposing two consecutive tuples from both streams meet at the point α, as shown in Figure 11, then their communication times in the distribution tree mostly overlap with each other because they are pushed to the distribution tree one after another. They travel together (using the FIFO strategy) to reach the targeted join core. As above, here, we also assume homogeneous join cores and communication costs within the distribution tree. Then, the average visiting latency of SplitJoin is given by:

$$VL_{avg} = (T_{CL} \times log_b k) + (\frac{w}{2} \times T_P) \quad (6)$$

Thus, the average visiting latency is also logarithmic in the number of join cores, compared to the linear order in [10].

7 Experimental Results

In this section, we experimentally evaluate our SplitJoin implementation. All experiments are performed on a 32-core system. Our system is a Dell PowerEdge R820 featuring 4 × Intel E5-4650 processors and 32 × 16GB DDR3 memory (RDIMM, 1600 MHz, Low Volt, Dual Rank, x4). We ran our benchmarks on Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86_64) installed on a Docker container [34] running on the same host OS.

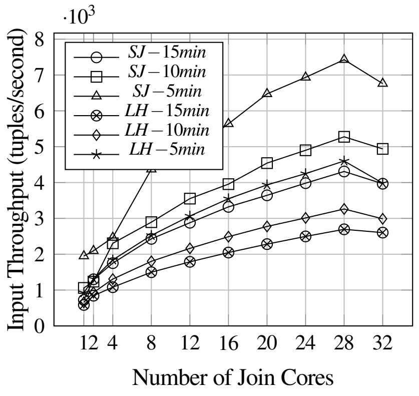

Figure 12: Throughput comparison.

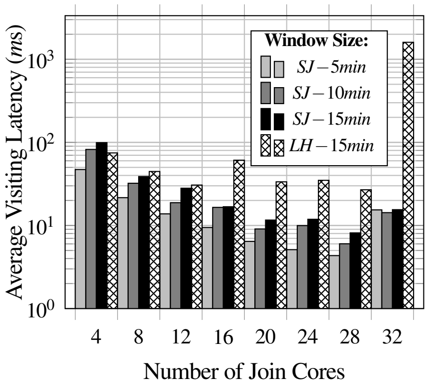

Figure 13: Visiting latency comparison.

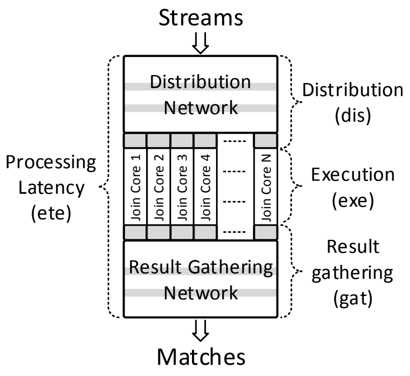

Figure 14: SplitJoin processing pipeline.

7.1 Experimental Setup

We adopted the benchmark used in recent stream join approaches [6, 9, 10]. In this benchmark two streams $R =$ `(x:int, y:float, z:char[20])` and S = `(a:int, b:float, c:double, d:bool)` are joined via the two-dimensional band join, as follows:

```
WHERE r.x BETWEEN s.a-10 AND s.a+10
  AND r.y BETWEEN s.b-10 AND s.b+10
```

In our evaluations, we used the low-latency (referred to as *LH vs.* SplitJoin (*SJ*)) and the original handshake join libraries that were kindly provided by the authors of [9, 10]. Also in line with related approaches, integers and floats were generated following a uniform distribution in the range of $1-10^4$, unless otherwise stated.

The results cover the end-to-end evaluation, including data distribution network, SplitJoin storage and processing cores, the result gathering network, and also the proposed punctuated ordering mechanism. The punctuation precision is based on the outer tuple ordering as shown in Figure 9, unless otherwise stated.

In our time-based window realization, we generated timestamps on-the-fly using the system call `clock_gettime()`. Using a synthetic timing mechanism, as we experimented, further improves the overall performance by about 15% by relieving the overhead incurred by system calls.

7.2 Performance & Scalability

We evaluate SplitJoin performance by measuring latency and throughput metrics as we scale the level of parallelism. In general, key factors that influence the stream join performance are how the input streams are flowing through the join cores and how the joined results are collected and flow to the output.

In Figure 12, we demonstrate throughput results of SplitJoin in comparison with [10]. As we scale the number of join cores, we observe that both solutions scale gracefully; however, SplitJoin outperforms the low-latency handshake join by up to 60% (comparison between 15-min sliding windows). Theoretically, the performance of both approaches should be similar, as both utilize all join cores in parallel to process incoming tuples. However, the core-to-core communication and mandatory expiry messages in low-latency handshake join (necessary for both time-based and count-based join versions) impose a noticeable penalty.

In Figure 12, we also observe how the two approaches perform for different time-based window sizes. When the join core count is 32, we observe a drop in performance in both of the approaches. This is due to the existence of extra threads to perform other (non-processing) tasks such as stream distribution and result gathering in case of SplitJoin, and tuple assignment, expiry message generation, and result gathering in case of the low-latency handshake join. Since our system has only 32 processing cores, by instantiating 32 join cores, the operating system is forced to perform context switches, resulting in system saturation and performance drop.

7.3 Latency Evaluations

In Figure 14, we present an abstract model of our end-to-end processing pipeline stages. The grayed parts show intermediate and pipeline buffers. In the measurements, we are reporting the latency of the distribution stage (*dis*), which also includes the time that tuples are waiting in the pipeline stage between the distribution and execution stages. The latency of the execution stage (*exe*) is the latency attributed to the time that it takes a tuple to pass through the processing and storage steps in the join core, which also includes the tuple expiration process for the time-based sliding window. The latency for the last stage includes the time that resulting tuples are waiting in the pipeline stage between the execution and result gathering stages and also the time for the *Merger* and the *Collector* units to bring them to the output of SplitJoin. Latency reports for these measurements plus the processing, end-to-end (*ete*), and latency of SplitJoin, for the time-based sliding window, are presented in Figure 15.

Processing Pipeline Stage Latency: In the distribution network, as we increase the number of join cores, incoming tuples are distributed between larger number of join cores instead of having to pile up in the pipeline buffer for fewer join cores. Therefore, increasing the number of join cores, inherently reduces the waiting time in the distribution stage as shown in Figure 15.

Since our evaluation system has only four processor sockets, the increase in the size of the distribution network has no significant effect on the performance except when the size of the sliding window is small. In the execution stage, the increase in the number of join cores for a given window size translates into smaller sub-windows for each join core and the latency also proportionally decreases.

Among our three pipeline stages, the result gathering network with punctuation ordering had the highest latency impact. This latency was mainly due to the waiting times of the *Mergers* on one of their input ports to receive a punctuation mark (*star*) before starting to read from their next port.

Visiting Latency: In Figure 13, we observe the average visiting latency ($T_{match} - max(t_r, t_s)$) for SplitJoin with 5, 10, and 15 minutes sliding windows, and low-latency handshake join with a 15 minutes sliding window for varying number of join cores. The t_r and t_s stand for initial timestamp of r and s tuples, respectively.

As we evaluated the average visiting latency (*cf.* Section 6), the latency increases logarithmically, $O(\log_b k)$, for SplitJoin as opposed to linearly, $O(k)$, for the low-latency handshake join [10]. By comparing the average visiting latency for the 15-min version of SplitJoin and low-latency handshake join, when we use four join cores, the latency is quite similar; however, once the number of join cores increases, the gap between SplitJoin and low-latency handshake join widens drastically by a factor of up to 3.3X (8.1*ms* *vs.* 26.8*ms* for 28 join cores).

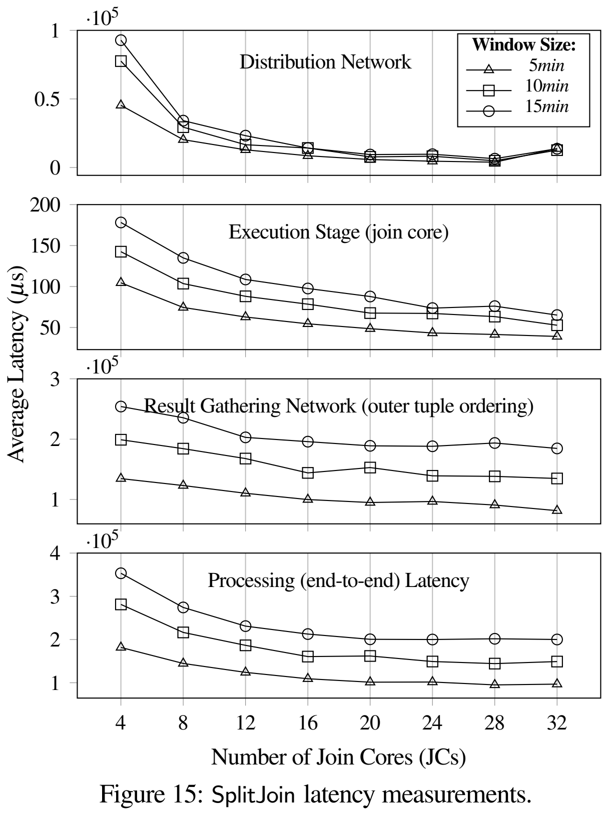

Figure 15: SplitJoin latency measurements.

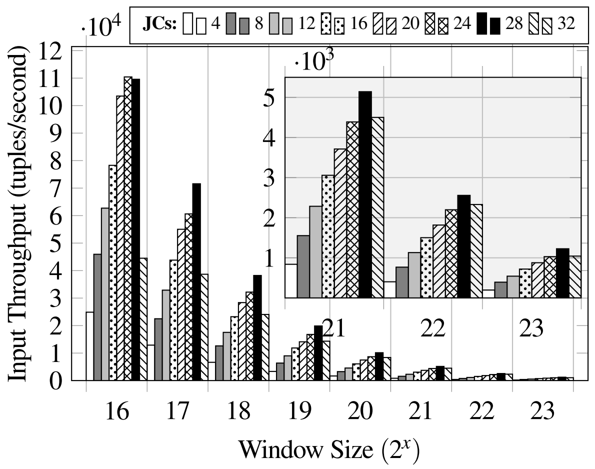

Figure 16: Count-based SplitJoin throughput.

We observe an increase in latency while reaching 32 join cores which is again due to the lack of enough resources for the other (non-processing) tasks. Since low-latency handshake join requires to perform additional costly tasks, such as emitting individual expiry message for each tuple, the resource contention shows a more significant impact on latency as seen when instantiating 32 join cores.

7.4 Count-based Sliding Window

Although the count-based and time-based versions of SplitJoin behave similarly, there are two key differences: (1) having no space allocated for timestamp values and no on-the-fly generation of timestamps through a costly system call and (2) having a fixed window size for count-based semantics as opposed to the time-based semantics where the window size varies depending on the incoming tuple rate. These differences result in roughly 20% improvement in performance for SplitJoin using a count-based instead of time-based sliding window. For example, SplitJoin instantiated with 28 join cores over a 15-min sliding window (shown in Figure 12) sustains an input rate of 4400 tuples/second, which roughly translates to window sizes of 2^{21} for each stream. But for the count-based window, if we set the window size to (2^{21}), SplitJoin can process up to 5200 tuples/second, as shown in Figure 16.

In the count-based results shown in Figure 16, we observe two effects: (1) larger window sizes result in fewer punctuation marks, assuming the same input throughput, since in the outer tuple ordering each join core produces one punctuation mark at the end of each tuple processing and (2) a larger sub-window per join core additionally increases the processing efficiency by reducing the impact of other (non-processing) pipeline stages. Based on these observations, doubling the window size while fixing the number of join cores reduces the processing throughput.

For each window size, by increasing the number of join cores (**JCs**), we observe a relative improvement in the

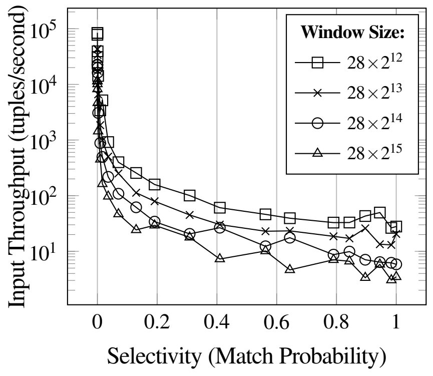

Figure 17: Selectivity effect on SplitJoin throughput (28 JCs).

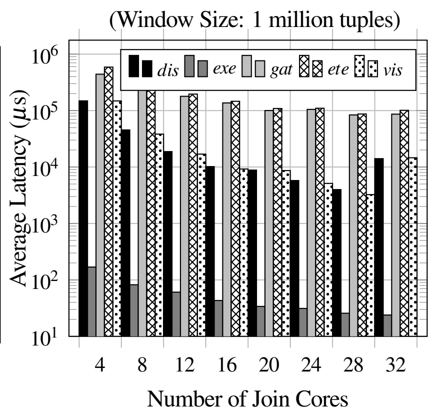

Figure 18: Count-based SplitJoin latency reports (uniform distribution:$1-10^5$).

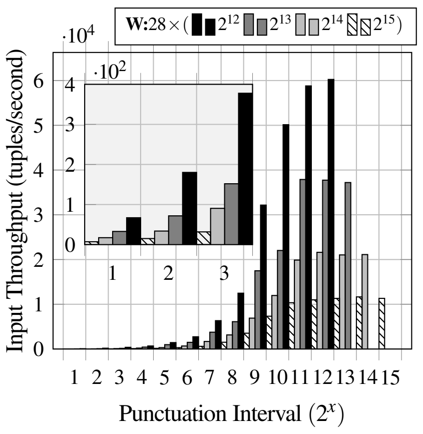

Figure 19: Ordering precision effect (28 JCs, uniform distribution:$1-10^4$).

throughput except when using 32 join cores. Over-utilizing system resources (*i.e.*, using 32 join cores) has more impact on the throughput for smaller window sizes. Larger windows keep join cores busier, thus, new tuples are processed after longer waits. This relieves other tasks (*i.e.*, distribution), reducing the effect of resource contention.

In Figure 18, we present the latency of the processing pipeline stages, the average processing latency (*ete*), and visiting (*vis*) latency for SplitJoin for the count-based sliding window. SplitJoin scales gracefully as we increase the number of join cores; in particular, using 28 join cores, the visiting latency is improved by more than 2.5X and 8.3X as compared to the time-based version of SplitJoin and the low-latency handshake join, respectively.

7.5 Effect of Selectivity

The selectivity (also called match probability) is one of the major factors affecting join performance. Often a low selectivity is assumed in most related work [6, 9, 10]. However, it is important to analyze the sensitivity of a join algorithm with respect to the selectivity in order to assess the generality of the approach.

The effect of varying the selectivity on the input throughput is illustrated in Figure 17. The key observation is that SplitJoin's latency scales reasonably, and it is robust to changes of selectivity, even for sliding windows as large as 28×2^{15} tuples.

7.6 Effect of Punctuation Precision

Figure 19 demonstrates the effect of the ordering precision on the processing performance. In this diagram, we utilize 28 join cores with varying sub-window sizes ($2^{12}-2^{15}$) per join core. The ordering precision starts from one punctuation per sub-window processing, referred to as *relaxed inner ordering*, and progressively increases the precision until one punctuation mark (*star*) is produced after each comparison (represented as 2^1 on the x-axis) within each sub-window, referred to as *strict inner ordering*.

The *relaxed inner ordering* is the same as the *strict outer ordering*. Therefore, the highest punctuation interval for each window size in Figure 19 represents the effect of *strict outer ordering* on the throughput for that window size.

As we increase the precision (*e.g.*, focusing on a sub-window size of 2^{15}), from $2^{11}-2^{15}$, its effect on the overall performance is negligible, since the number of punctuations produced per each incoming tuple in each join core is relatively low (*i.e.*, 1, 2, 4, 8, and 16 punctuations, respectively) compared to the sub-window size which is 2^{15}. However, as we continue to increase the precision from the interval 2^{10} down to 2^1, the number of punctuations becomes comparable to the sub-window size for each join core, and as expected, negatively affects the performance of SplitJoin. This highlights the importance of balancing ordering precision versus overall performance. In fact, since the precision is adjustable, to achieve a desired throughput, SplitJoin could adaptively adjust the precision interval to achieve a sweet spot between the ordering precision and the sustainable input throughput.

8 Conclusions

We present SplitJoin, a novel stream join that introduces two unique properties that distinguish it from existing work. First, SplitJoin exhibits a scalable architecture that splits the join computation into two independent "storing" and "processing" steps that can be parallelized using a coordination-free protocol to achieve low-latency join processing. Second, SplitJoin introduces a simplified top-down, flow-oriented join processing that eliminates complex concurrency logic for avoiding race conditions while satisfying input stream ordering semantics. We further propose scalable distribution- and collection-trees for input stream propagation and output join result gathering, respectively. Lastly, we propose a relaxed adjustable punctuation to guarantee join result ordering and to provide an effective mechanism to balance the trade-offs between the ordering precision and the overall join throughput.

Acknowledgment

This research was supported by the Alexander von Humboldt Foundation.

References

[1] SRIVASTAVA, D., GOLAB, L., GREER, R., JOHNSON, T., SEIDEL, J., SHKAPENYUK, V., SPATSCHECK, O., AND YATES, J. Enabling real time data analysis. VLDB'10.

[2] SADOGHI, M., JACOBSEN, H.-A., LABRECQUE, M., SHUM, W., AND SINGH, H. Efficient event processing through reconfigurable hardware for algorithmic trading. VLDB'10.

[3] CRANOR, C., JOHNSON, T., AND SPATASCHEK, O. Gigascope: a stream database for network applications. SIGMOD'03.

[4] FONTOURA, M., SADANANDAN, S., SHANMUGASUNDARAM, J., VASSILVITSKI, S., VEE, E., VENKATESAN, S., AND ZIEN, J. Efficiently evaluating complex Boolean expressions. SIGMOD'10.

[5] GEDIK, B., BORDAWEKAR, R. R., AND YU, P. S. CellSort: high performance sorting on the cell processor. VLDB'07.

[6] GEDIK, B., BORDAWEKAR, R. R., AND YU, P. S. CellJoin: A parallel stream join operator for the cell processor. VLDBJ'09.

[7] MARGARA, A., AND CUGOLA, G. High performance content-based matching using GPUs. DEBS'11.

[8] TUMEO, A., VILLA, O., AND SCIUTO, D. Efficient pattern matching on GPUs for intrusion detection systems. CF'10.

[9] TEUBNER, J., AND MUELLER, R. How soccer players would do stream joins. SIGMOD'11.

[10] ROY, P., TEUBNER, J., AND GEMULLA, R. Low-latency handshake join. VLDB'14.

[11] BLANAS, S., LI, Y., AND PATEL, J. M. Design and evaluation of main memory hash join algorithms for multi-core CPUs. SIGMOD'11.

[12] TEUBNER, J., ALONSO, G., BALKESEN, C., AND OZSU, M. T. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. ICDE'13.

[13] LEIS, V., BONCZ, P., KEMPER, A., AND NEUMANN, T. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD'14.

[14] MUELLER, R., TEUBNER, J., AND ALONSO, G. Data processing on FPGAs. VLDB'09.

[15] HAGIESCU, A., WONG, W.-F., BACON, D., AND RABBAH, R. A computing origami: Folding streams in FPGAs. DAC'09.

[16] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. Flexible query processor on FPGAs. VLDB'13.

[17] MUELLER, R., TEUBNER, J., AND ALONSO, G. Streams on wires: a query compiler for FPGAs. VLDB'09.

[18] WOODS, L., TEUBNER, J., AND ALONSO, G. Complex event detection at wire speed with FPGAs. VLDB'10.

[19] SADOGHI, M., JAVED, R., TARAFDAR, N., SINGH, H., PALANIAPPAN, R., AND JACOBSEN, H.-A. Multi-query stream processing on FPGAs. ICDE'12.

[20] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. Configurable hardware-based streaming architecture using online programmable-blocks. ICDE'15.

[21] KANG, J., NAUGHTON, J., AND VIGLAS, S. Evaluating window joins over unbounded streams. ICDE'03.

[22] LIN, Q., OOI, B. C., WANG, Z., AND YU, C. Scalable distributed stream join processing. SIGMOD'15.

[23] CHANDRASEKARAN, S., ET AL. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR'03.

[24] CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. NiagaraCQ: A scalable continuous query system for internet databases. SIGMOD'00.

[25] ABADI, D. J., ET AL. The design of the Borealis stream processing engine. CIDR'05.

[26] WU, L., BARKER, R. J., KIM, M. A., AND ROSS, K. A. Navigating big data with high-throughput, energy-efficient data partitioning. ISCA'13.

[27] XIE, J., AND YANG, J. A survey of join processing in data streams. In *Data Streams*, C. Aggarwal, Ed., vol. 31 of *Advances in Database Systems*. Springer US, 2007.

[28] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., AND WHITTLE, S. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. VLDB'15.

[29] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., TANEJA, S. Twitter Heron: Stream processing at scale. SIGMOD'15.

[30] Apache Storm. http://storm.apache.org.

[31] SOURDIS, I., AND PNEVMATIKATOS, D. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion. FPL'03.

[32] BORDAWEKAR, R. R., AND SADOGHI, M. Accelerating database workloads by software-hardware-system co-design. ICDE'16.

[33] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. The FQP vision: Flexible query processing on a reconfigurable computing fabric. SIGMOD'15.

[34] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. Linux J'14.