A Scalable Circular Pipeline Design for Multi-Way Stream Joins in Hardware

Mohammadreza Najafi†, Mohammad Sadoghi‡, Hans-Arno Jacobsen†
†*Technical University of Munich*
‡*University of California, Davis*

***Abstract*—Efficient real-time analytics are an integral part of a growing number of data management applications such as computational targeted advertising, algorithmic trading, and Internet of Things. In this paper, we primarily focus on accelerating stream joins, arguably one of the most commonly used and resource-intensive operators in stream processing. We propose a scalable circular pipeline design (Circular-MJ) in hardware to orchestrate multi-way join while minimizing data flow disruption. In this circular design, each new tuple (given its origin stream) starts its processing from a specific join core and passes through all respective join cores in a pipeline sequence to produce final results. We further present a novel two-stage pipeline stream join (Stashed-MJ) that uses a best-effort buffering technique (stash) to maintain intermediate results. In a case that an overwrite is detected in the stash, our design automatically resorts to recomputing intermediate results. Our experimental results demonstrate a linear throughput scaling with respect to the number of execution units in hardware.**

I. Introduction

In recent years, there has been an increasing interest in data stream management systems which encompasses a wide range of applications such as real-time data analytics, targeted advertising, data mining, and Internet of Things. The common pattern among these applications is a predefined set of streaming queries (e.g., ad campaigns or trading strategies) and unbounded event streams of incoming data (e.g., user profiles or stock feeds) that must be processed against queries in real-time. These latency-sensitive and throughput-intensive applications have opened the demand for accelerating data management operations in general and stream processing in particular.

Considering the crucial role of joins as resource-intensive operators in relational databases, it is not of a surprise that stream joins have also been the focus of much research on data streams [1], [2], [3], [4], [5], [6], [7], [8]. For example, consider TPC-H [9] where 20 queries (out of 22) contain join operator while 12 of them use multi-way joins some up to 7 joins. However, the importance of joins is no longer limited to only the classical relational setting. The emergence of Internet of Things (IoT) has introduced a wide wave of applications that rely on sensing, gathering, and processing data from an increasingly large number of connected devices.

In stream join processing, software platforms offer flexible communication where we see anycast and multicast connections between internal components without a significant drop in performance. As an example, assume a system with four internal components of A, B, C, and D where a point-to-point connection between them could build a data-path similar to $A \rightarrow B \rightarrow C \rightarrow D$. In a software platform establishing communication i.e., $A \rightarrow D$ and $B \rightarrow D$ is not detrimental especially given the flexible shared memory hierarchy. State-of-art software approaches in a multi-way stream join benefit from this

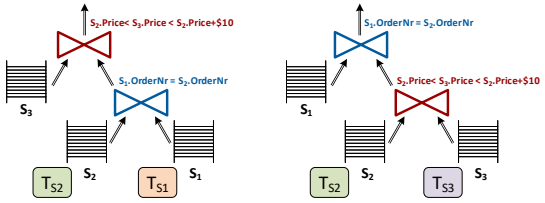


Fig. 1: Reordering operators in multi-way stream joins.

provided flexibility that leads to an unstructured[1] architecture. However, on hardware, it is instrumental to predetermine and plan the necessary communication channels otherwise designs' performance, complexity, and cost rapidly increase since all communications have to be exclusively realized using physical wiring. This motivates complete rethinking of the hardware design rather than simply relying on re-implementation of available software solutions. On the other hand, the execution of continuous queries (i.e., repetitive tasks) on potentially unbounded streams using the finite-window semantics offer a unique opportunity and make it suitable for hardware acceleration. A hardware solution presents negligible, if any, gains when executing an operation only once versus its competing software variant; however, when this operation repeats many times, the amortized gains grow far beyond that of a software.

Multi-Way Stream Join: Past hardware solutions [3], [4], [10], [11], [12], [13], [14] focus on a join between two streams while practical queries often go beyond joining two streams. Notice it is non-trivial to build multi-way join operators by cascading operators designed for two streams as each new tuple, depending on its origin, requires its own order of join operators for the processing. Therefore, it remains a major challenge to design multi-way stream joins in hardware due to excessive cost and penalty of flexible communications.

In the first part of this work, we propose a novel circular pipeline architecture for multi-way stream join (Circular-MJ) that uses a dedicated stage for each stream sliding window. This inherently limits data dependency between stages by using results of each stage as input for the next stage, leading to a scalable architecture that is centered around direct neighbor-to-neighbor communication. We use two fundamental steps to reshape the problem of multi-way join to design a scalable hardware architecture. First, we change the problem of unstructured multi-way stream join designs to a join reordering problem in a structured point-to-point design. Second, we eliminate the join operator reordering problem by moving the reordering task to tuple insertion circuitry using a pipelined distribution chain.

Multi-Way Stream Join Optimization with Stash: Depending on streams' characteristics and join operators'

[1]We define unstructured architecture as an arbitrary point-to-point communication (or other classes of communication pattern such as anycast) among any processing nodes within the system.

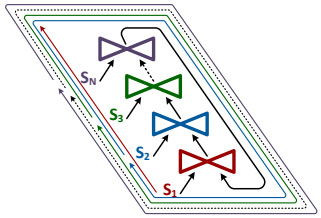


Fig. 2: Multi-way stream joins with circular design.

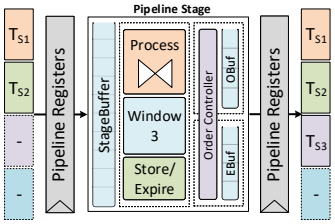


Fig. 3: Intermediate result generation in a pipeline stage.

properties, materializing intermediate results[2] in a buffer may provide performance improvements by avoiding recomputation of already processed tuples. However, in the introduced Circular-MJ, the processing engine for each stream is placed in a separate stage and accessing this buffer from two independent pipeline stages imposes multiple resource sharing challenges in a hardware realization. First, having a shared unit between two stages violates the main concept of pipeline design, which is the separation of concerns. Second, this sharing can result in race conditions between the storage and processing of two tuples at different stages that require expensive stalls in the pipeline to address.

In the second part of this work, we propose a custom two-stage (for three or more streams) pipeline (referred to as Stashed-MJ) including a stash[3]. The novelty of our approach is to benefit from the reduction in the number of pipeline stages in the favor of better utilization of available processing units and avoiding recomputation of already processed data. Furthermore, the processing unit in the first stage operates on two windows (that are also connected to the stash) but not at the same time, which also eliminates the resource sharing challenges.

In this paper, we make the following contributions:

A) Propose a scalable multi-way stream join (Circular-MJ) on hardware that is built on a circular chain of dedicated stages (one per stream) and benefits from pipeline parallelism.

B) Present a novel two-stage pipeline (Stashed-MJ) that benefits from a stash (intermediate results buffer) to accelerate processing throughput.

II. Multi-Way Stream Join

A conventional join operates on tuples originating from two sources. Naturally, we should be able to cascade the join operators to support more than two sources (streams). However, avoiding arbitrary (unstructured) communications between processing components, as a crucial property for hardware design, introduces the challenge of real-time join operator reordering as demonstrated in Figure 1. Here tuples from S_1 and S_2 streams keep the order of join operators intact (left figure), while a new tuple from S_3 stream mandates the operator reordering (right figure). Without the reordering, we have to recompute all intermediate join results between all existing tuples in the sliding windows of S_1 and S_2 which is not feasible due to the size and complexity of this processing. The reordering challenge is exacerbated when dealing with a hardware system, where changes in data-path and control circuitry of a design, especially as it is scaled up (i.e., in the number of streams), have severe effects on design complexity, performance, and cost of the system.

[2]Outcomes of each join operator, except the last one in a join tree which emits the final results, are referred to as intermediate results.

[3]We refer to intermediate result buffer with additional control circuitries as stash.

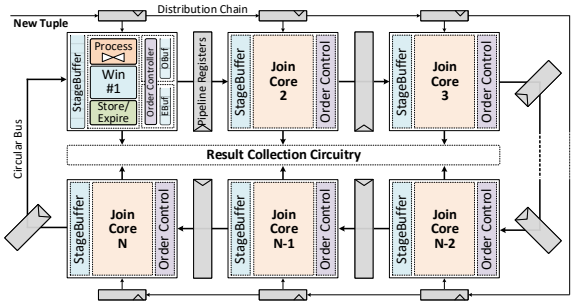


Fig. 4: Circular-MJ architecture.

Each new tuple insertion into the multi-way stream join updates its sliding window and subsequently may produce new intermediate results. Without materializing the intermediate results, we need to sequentially cascade the join operators and feed each new tuple always from the bottom (input port) of the cascaded architecture. Each new tuple and subsequently its intermediate results pass through all join operators for processing which leads to a right-deep join tree architecture.

A. Multi-Way Join with Circular Pipeline

Designing a hardware based on the join reordering poses a scalability issue due to the required crossbar[4] connection between processing units (join operators) and sliding windows. To address this issue, we need to fix the order of join operators which hardwires each sliding window to only one processing unit. This eliminates the need for a crossbar. To fix each join operator's location on the right-deep join tree, we propose a circular data-path design which connects all operators together, as shown in Figure 2. In this design, each join operator is connected to only one sliding window with two entries. Each operator receives its new tuples, determined based on their origin, from its right entry. The left entries are placed in the closed circular path which carries intermediate results from a join operator to the next one. Resulting tuples are emitted after processing a new tuple in exactly $N-1$ operators, where we have N streams. The remaining operator is responsible to store the new tuple in its sliding window.

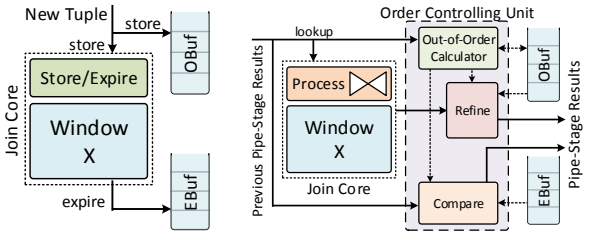
Using this design (Figure 2), we propose a scalable circular pipeline for multi-way stream join in hardware (referred to as Circular-MJ) that is shown in Figure 4. This pipeline has the same number of stages as input streams. Each stage is placed between two isolating sets of registers and is responsible to process a new tuple against a specific sliding window. In case the window in a stage belongs to the current tuple's origin, store and expiration tasks are performed instead of the processing.

B. Circular Pipeline Design Rationale

The key factor that has heavily influenced our design is having an independent operation on each window assuming we already have intermediate results from another join operator(s). Therefore, we design N stages, each responsible for processing, storage, and expiration operations on a single sliding window. To handle data transfer between stages in a scalable manner, we arrange them in a circular architecture in such a way that intermediate results of each stage are fed to another stage as input.

The key intuition of our circular design is that instead of adapting the order of join operations to incoming tuples,

[4]A type of connection which provides the possibility for every input to access to all output ports.



(a) *Store* (b) *Lookup*

Fig. 5: Order control unit architecture.

we adapt the insertion location for incoming tuples using a pipelined distribution chain. In other words, instead of having a single entry and reordering join operators we change the architecture to have an adaptive entry to fix the join operators order; and respectively, to hardwire each sliding window to a separate processing unit (Figure 4).

Our circular design differs from a conventional pipeline in three ways: (1) A circular bus that passes through all stages within pipeline that relies exclusively on direct neighbor-to-neighbor communication. (2) Arbitrary access to any stage in the pipeline as opposed to physically fixing each stream to one join operator in the pipeline. (3) Arbitrary output collection from all stages that is necessary to offload the final results while sustaining high throughput.

C. Circular-MJ Architecture

Our circular pipeline consists of N stages. Each stage is surrounded by two isolating sets of pipeline registers that are connected to other stages using the *circular bus* as shown in Figure 4. This bus provides a processing path that encompasses all pipeline stages. Each stage contains three main components: (1) a StageBuffer, (2) a Join Core, and (3) an Order Control Unit. To feed the incoming tuples to the stages, we use a pipelined chain of registers (referred to as *distribution chain*) that carries new tuples to their corresponding stage. The purpose of the distribution chain is to keep the circular pipeline at full utilization and maximize processing throughput.

(1) **StageBuffer** collects intermediate results from a previous stage and feeds them one-by-one to its stage's join core. The existence of this buffer is necessary to prevent deadlocks caused by the joint burst of intermediate results and new tuples. A deadlock may occur due to a low selectivity (high match probabilities[5]) in multiple consecutive join operators in the deep join tree that leads to the generation of many intermediate results.

In the deadlock scenario, all StageBuffers are full and each stage is waiting for the next stage to consume some of the intermediate results in its buffer for further processing. Looking at this scenario from the perspective of i^{th} stage that is waiting for $(i+1)^{th}$ stage to consume some of the intermediate results in its StageBuffer. Then, given N streams, the n^{th} stage waits for the 1^{st} stage and this dependency reaches to the $(i-1)^{th}$ stage which is waiting for the i^{th} stage since the pipeline stages are arranged in a circular architecture. Therefore, the i^{th} stage is waiting for itself to continue the processing which translates to a deadlock. Using a StageBuffer at the entry of each stage prevents deadlocks from happening by providing extra space to store intermediate results. This

[5]The probability that any two consecutive tuples (each from one of streams) satisfy a join condition. Thus, as *selectivity* increases, the *match probability (mp)* decreases.

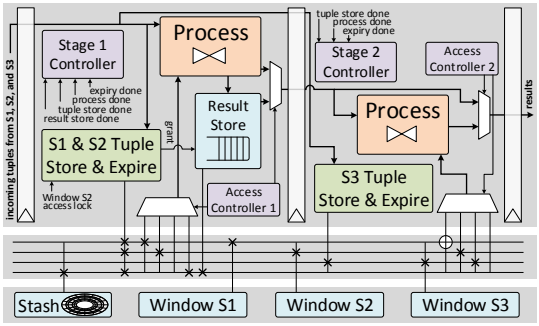


Fig. 6: Stashed-MJ architecture.

way the i^{th} stage can push its produced intermediate results without waiting for the next stage to consume them.

Furthermore, we give priority to intermediate results over new tuples, which further reduces the chance of a deadlock. This means, when there is a choice between processing an intermediate result or a new tuple, a stage controller picks the former to reduce the StageBuffer's fill-ratio. Additionally, to fully prevent deadlocks, we drop an intermediate result when the StageBuffer fill-ratio is beyond a specified threshold.

(2) **Order Control Unit** is responsible for the correct execution order. The out-of-order tuple insertion into the processing pipeline by the distribution chain can lead to incorrect (match by consecutive tuples) or missed results (missing prematurely expired tuples). This unit gets involved in two tasks of a new tuple store and lookup, as shown in Figure 5. This unit has an OBuf that stores newly received tuples from the distribution chain and an EBuf that stores expired tuples from the sliding window. These two buffers are the same size as the number of streams (pipeline stages). The order control unit has an *Out-of-Order Calculator* which counts the number of tuples belonging to its stage's sliding window which are stored in this window prematurely[6]. To avoid incorrect matches with prematurely stored tuples, the order control unit uses a *Refine* component to drop them. To avoid missing matches with prematurely expired tuples, a *Compute* component performs an additional comparison with the same number as the prematurely stored tuples, with the last expired tuples in the EBuf.

(3) **Join Core** contains its dedicated stream's sliding window in addition to the processing, storage, and expiry components that operate on this window. The sliding window can be count-based or time-based, and the joining algorithm can encompass different approaches from nested-loop, used for general joins, to hash-based, used for equi-joins.

III. Multi-Way Stream Join with Stash

We now turn our attention to another key dimension of multi-way join architecture by introducing Stashed-MJ, a novel buffering and an improved pipelining that consists of two key ideas. The first property of Stashed-MJ design is the integration of a buffer, referred to as *stash*, to materialize intermediate results in order to substantially improve the throughput by avoiding recomputation of previously processed tuples. The second property is centered around improving resource utilization. In Circular-MJ, only $N-1$ stages out of N stages perform the processing such that, for each stream, one stage is always occupied with the storage and expiry tasks. Therefore, there is always one unused processing unit in the circular pipeline.

[6]Earlier than their order in correspondence with the current tuple under processing.

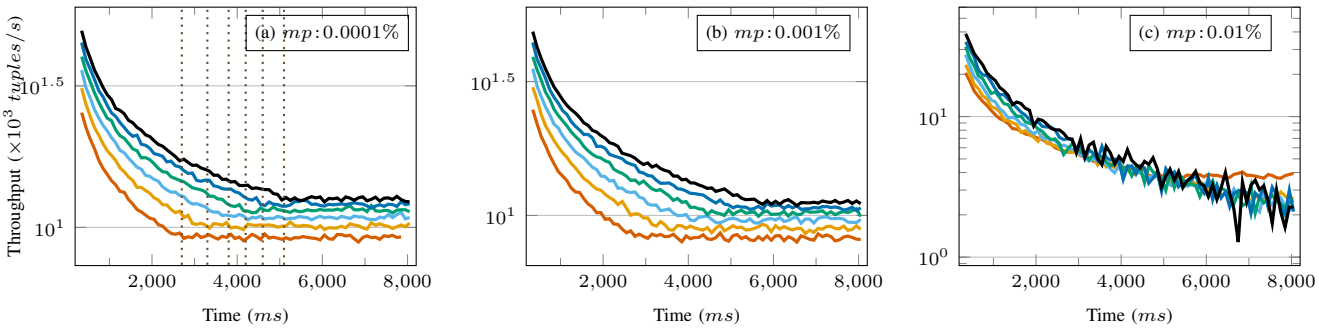


Fig. 7: Stream count effect on input throughput (w:2^{14}, $stream(stage)$ $count$:3—,4—,5—,6—,7—,8—).

Integration of the stash into Circular-MJ (Figure 4) imposes an important design challenge because we are now forced to share the buffer among more than one processing units that are placed into two or more separate pipeline stages. This raises two concerns. First, having a shared block between two pipeline stages that could be accessed frequently and in parallel violates the main concept of the circular pipeline design that is based on separation of concerns and a strictly one-way neighbor-to-neighbor communication. Second, this sharing can result in race conditions between concurrent processing of a new tuple and storing the matched results of an earlier tuple, which now demands expensive pipeline stalls for coordination.

In Circular-MJ, we utilize three pipeline stages for a 3-way stream join operation. For every tuple insertion, two of these stages process this tuple against the sliding window of other streams while the remaining stage is responsible for storing this tuple in its stream's sliding window and expiring the oldest tuple from it. There are two key insights that guide our new Stashed-MJ design: (1) storage and expiration operations are relatively less costly compared to processing (especially in nested-loop stream join), and (2) storage and expiration are performed on a separate sliding window than the windows that are used for processing. By exploiting these insights, we reduce the number of pipeline stages to two by performing storage and expiration in parallel with the processing in the first pipeline stage. As a result, the processing unit in the first stage has to operate on two sliding windows, depending on newly received tuples' origin, but not at the same time. This provides us with the opportunity to offload the processing operation of two streams that are involved in updating the stash onto this stage, which eliminates the sharing challenge as shown in Figure 6.

IV. Experimental Results

We have realized our multi-way stream join pipeline (Circular-MJ) and optimized pipeline with stash (Stashed-MJ) in VHDL. For evaluations we used `Questa Advanced Simulator` to extract cycle accurate measurements (with a clock frequency of 100MHz), guaranteeing the same performance for the actual hardware. We employed input streams that consist of 64-bit (32-bit key and 32-bit value) tuples that are joined against other streams sliding windows.

To measure the effect of stream count on input throughput we use join conditions with high selectivity (low match probability) to approach the maximum sustainable throughput, presented in Figure 7a. As expected from the pipelining parallelism, increase in the number of pipeline stages linearly improves the processing throughput. This shows the effectiveness of the distribution chain since it has been able to keep the pipeline stages busy which is the key factor in pipelining parallelism. On the left side of this figure we first observe a warm-up phase where we have a super linear reduction in the input throughput as sliding windows get filled. The vertical dash lines specify the end of the warm-up phase for pipelines with a different number of streams.

Figures 7b and 7c present input throughput for lower selectivities. We only observe a small reduction in the input throughput in Figure 7b while the measurements for match probability (mp) of 0.01%, Figure 7c, show a much lower and also sporadic throughput readings. Increasing the match probability leads to more intermediate results in each pipeline stage which enforce next stages to process them instead of receiving new tuples. In a smaller pipeline, intermediate results require less number of stages to construct final results which are the reason for smaller throughput drop for pipelines with lower number of streams in Figure 7c.

V. Conclusions

We have focused on hardware acceleration of multi-way stream joins. In particular, we presented Circular-MJ, a novel circular pipeline architecture for realizing multi-way stream join that eliminates the need to re-arrange the order of join operators and avoids arbitrary point-to-point communication among custom join cores. We further expanded our multi-way join design, referred to Stashed-MJ, to efficiently cache intermediate results in order to avoid recomputing the already processed data.

References

[1] C. Koch, "Incremental query evaluation in a ring of databases," ser. PODS'10.
[2] J. Kang, J. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," *ICDE*, 2003.
[3] B. Gedik, R. R. Bordawekar, and P. S. Yu, "CellJoin: A parallel stream join operator for the cell processor," *VLDBJ*, 2009.
[4] J. Teubner and R. Mueller, "How soccer players would do stream joins," *SIGMOD*, 2011.
[5] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha, "DBToaster: higher-order delta processing for dynamic, frequently fresh views," *VLDBJ'14*.
[6] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," *Big Data*, 2015.
[7] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, "Scalable distributed stream join processing," *SIGMOD*, 2015.
[8] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision," *USENIX ATC*, 2016.
[9] T. P. P. Council, "TPC-H benchmark specification," *Published at http://www.tcp.org/hspec.html*, 2008.
[10] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H. A. Jacobsen, "Multi-query stream processing on FPGAs," *ICDE*, 2012.
[11] M. Najafi, K. Zhang, M. Sadoghi, and H.-A. Jacobsen, "Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations," *ICDCS*, 2017.
[12] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "Design and implementation of a handshake join architecture on FPGA," *IEICE*, 2012.
[13] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Configurable hardware-based streaming architecture using online programmable-blocks," *ICDE*, 2015.
[14] C. Kritikakis, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos, "An FPGA-based high-throughput stream join architecture," *FPL*, 2016.