ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join

Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas
Chalmers University of Technology, Gothenburg, Sweden
$\{$*vinmas, ioaniko, ptrianta, tsigas*$\}$*@chalmers.se*

***Abstract*—The inherently large and varying volumes of data generated to facilitate autonomous functionality in large scale cyber-physical systems demand near real-time processing of data streams, often as close to the sensing devices as possible. In this context, data streaming is imperative for data-intensive processing infrastructures. Stream joins, the streaming counterpart of database joins, compare tuples coming from different streams and constitute one of the most important and expensive data streaming operators. Dictated by the needs of big data streaming analytics, algorithmic implementations of stream joins have to be capable of efficiently processing bursty and rate-varying data streams in a deterministic and skew-resilient fashion. To leverage the design of modern multicore architectures, scalability and parallelism need to be addressed also in the algorithmic design.**

In this paper we present ScaleJoin, an algorithmic construction for deterministic and parallel stream joins that guarantees all the above properties, thus filling in a gap in the existing state-of-the art. Key to the novelty of ScaleJoin is a new data structure, Scalegate, and its lock-free implementation. ScaleGate facilitates concurrent data exchange and balances independent actions among processing threads; it also enables fine-grain parallelism while providing the necessary synchronization for deterministic processing. As a result, it allows ScaleJoin to run on an arbitrary number of processing threads that can evenly share the overall comparisons run in parallel and achieve high processing throughput and low processing latency. As we show, ScaleJoin not only guarantees deterministic, disjoint and skew-resilient parallelism, but also achieves higher throughput than state-of-the-art parallel stream joins.

I. Introduction

The world-wide adoption of large cyber-physical systems (e.g., smart grids, smart vehicular networks or enhanced medical systems) demands for near real-time processing of continuous streams of data [10]. In this context, the distributed and parallel analysis enabled by data streaming overcomes the limitations of store-than-process approaches. In this computing paradigm, graphs of stream operators are employed to process data in an online fashion.

Stream joins are among the most important and expensive stream operators [18], [16], [1]. In contrast to their database counterparts, they compare tuples coming from data streams rather than relations. Due to the unbounded nature of data streams, such comparisons are performed on portions of the most recent tuples, referred to as *windows*. The design and implementation of *high throughput* and *low latency* parallel stream joins is challenging because of their high computational cost [1]. In the literature, both shared-nothing [9], [1] and shared-memory [6], [18], [16] parallelization techniques have been proposed. The former allows for parallel stream joins to scale out in multi-node deployments while the latter has been shown to successfully *scale up* performance within individual nodes.

As emphasized by Gibbons [7], scaling both out and up is crucial to effectively improve performance by orders of magnitude. Nevertheless, state-of-the-art shared-memory parallel stream joins suffer from two main shortcomings that limit their adoption in multi-node parallel streaming applications. Specifically, they (i) assume tuples to be delivered by exactly two input streams (while practice demands to deal with arbitrary numbers of streams of tuples generated, for instance, by other parallel stream operators) or (ii) do not guarantee deterministic processing (crucial in sensitive applications as clickstream analysis, for which reporting wrong revenue to investors would cause money loss [1]).

ScaleJoin: A new parallelization perspective

Motivated by the aforementioned limitations, we aim at the design and implementation of a new shared-memory parallel stream join that: (1) is able to process tuples delivered by arbitrary numbers of input streams, (2) guarantees deterministic processing, and (3) is scalable and provides high-throughput and low-latency through disjoint and skew-resilient parallelism (cf. definition in Section IV-C).

Uniquely compared to previous work, we show how crucial it is, in order to meet these challenges, to focus on the underlying data structures of parallel stream joins. Through non-blocking and consistent synchronization it is possible not only to boost parallelism, but also to bridge the gap between existing shared-nothing and shared-memory parallel data streaming applications. Quoting from Michael [15], *"the choice of data structures is one of the most important decisions in designing a non-blocking environment."*

We introduce a new abstract data type, *ScaleGate*, that distills a minimal interface for satisfying the aforementioned determinism and parallelism requirements. We also provide a concurrent algorithm that implements this interface and allows data exchange and synchronization while guaranteeing determinism. For simplicity in the rest of this paper, unless otherwise mentioned, we refer to both the abstract data type and the data structure implementation as *ScaleGate*. Building on *ScaleGate*, we introduce *ScaleJoin*, which allows for the parallel execution of an arbitrary number n of processing threads (each running its share of the overall comparisons in parallel). A summary of our results is listed below:

1) After introducing a concise definition of deterministic processing for parallel stream joins, we prove deterministic processing for *ScaleJoin*.
2) By properly designing and implementing the underlying data structures in *ScaleJoin*, we balance and limit unnecessary synchronization overheads, thus ensuring more time for threads to run comparisons.
3) We provide the algorithmic implementation of *ScaleGate* through lock-free synchronization, along with its safety- and progress-guarantees proofs. *ScaleGate* allows for fine-grain interleaving of thread executions (i.e., enhanced parallelism) and ensures system-wide progress, enabling high scalability across varying multiprocessor architectures and for arbitrary number of input streams.
4) We address the disjoint parallelism and skew-resilience challenges by relying on *ScaleGate*, the parallel and concurrent coordinator whose instances feed the n processing threads and collect their output tuples. We provide an extensive experimental study, covering a broad range of setups.

ScaleJoin achieves high-throughput and low-latency processing and (i) is not bounded to language- or hardware-specific optimizations (e.g., SIMD instructions [6], [18], [16]); (ii) is architecture-independent (e.g., differently from CellJoin [6]); and (iii) allows for optimization techniques (e.g., equi-joins or band-joins) to be easily leveraged.

By focusing on the concurrent access to the data structures of parallel stream operators, we show a new way for benefits and paths to explore in the research for parallel streaming applications. To provide further evidence of such benefits, we include in our evaluation a discussion relating, in terms of throughput and latency, *ScaleJoin* and Handshake join [18], [16], the fastest stream join proposed in the literature.

The rest of the paper is organized as follows. Section II overviews the join semantics. Section III focuses on deterministic execution for both sequential and parallel implementations. Section IV introduces the *ScaleGate* abstract data type and *ScaleJoin*'s architecture. Detailed descriptions of the algorithmic implementations (including proofs of correctness and deterministic processing) are presented in Section V. We evaluate *ScaleJoin* in Section VI, discuss related work in Section VII and conclude in Section VIII

II. Preliminaries

We follow the description and semantics of stream joins in related literature (e.g., [6], [18], [9]), presented here for self-containment. A stream is an unbounded sequence of tuples $t_0,t_1,\ldots$ sharing the schema $\langle ts,A_1,\ldots,A_n\rangle$. Given tuple t, attribute $t.ts$ represents its creation timestamp while attributes $A_1,\ldots,A_n$ are application-related. Tuples in a stream are considered to be in timestamp order.

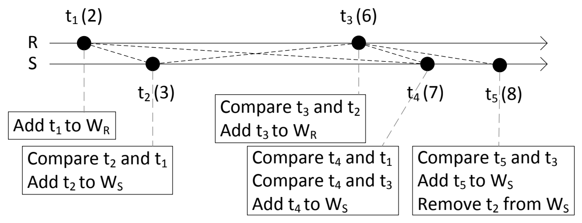


Figure 1: Procedure steps for a sample sequence of tuples ($\texttt{WS}$=4 time units). For each tuple, the figure shows its timestamp (in parenthesis) and the resulting procedure steps.

Stream joins compare tuples received from two *logical* streams, R and S, using *predicate* P. While defining two logical input streams, R and S tuples might be delivered by arbitrary numbers of *physical* streams, each delivering tuples in timestamp order [9] (e.g., by the physical streams produced by other parallel stream operators, as discussed in Section I). In the remainder, we often use the term stream without specifying whether it is logical or physical, since it can be deduced by the context.

Since streams are unbounded, tuples from each stream are compared only with a portion (*window*) of the opposite stream. We focus on *time-based windows* of $\texttt{WS}$ time units, that contain all tuples $\{t|t'.ts - t.ts \leq \texttt{WS}\}$, where t' is the latest received tuple in the respective stream[1]. Nevertheless, our parallelization technique can be easily extended to *tuple-based windows*, maintaining a fixed amount of the last $\texttt{WS}$ received tuples. Whenever $P(t_R,t_S)$ holds for tuples $t_R \in R$ and $t_S \in S$, an output tuple t_O is produced combining t_R and t_S and setting $t_O.ts = max(t_R.ts,t_S.ts)$.

The semantics of the stream join are commonly implemented as the *three-step procedure* proposed by Kang et al. [13]. W_R and W_S being the windows maintaining R and S tuples, respectively, the procedure applied for each incoming tuple $t_R \in R$ (symmetric for tuples $t_S \in S$) is:

1) *Compare* t_R with all $t_S \in W_S$.
2) *Add* t_R to W_R.
3) *Remove* all $t_i \in W_R : t_i.ts < t_R.ts - \texttt{WS}$.

Figure 1 presents the procedure's steps for a sample sequence of tuples. In the example, $\texttt{WS}$ is set to 4 time units.

Computational cost of stream joins The high computational cost of stream joins motivates the intensive research efforts for its parallelization. If we assume that both R and S receive T tuples per time unit, both W_R and W_S maintain $T \times \texttt{WS}$ tuples on average. Since each tuple from R is matched with the ones in W_S (and vice versa), the *average number of comparisons per time unit* is $2 \times \texttt{WS} \times T^2$. That is, if $WS = 10$ minutes and $T = 500$ tuples/second, a stream join must run 300 million comparisons/second on average.

[1]The notion of time in this context refers to tuples' timestamps, not to the physical clock of the system processing them.

III. Deterministic processing

The three-step procedure specifies how incoming tuples are compared and how windows evolve, but does not specify the order in which incoming tuples should be processed. Nonetheless, as we show in the following, the comparisons run by a stream join actually depend on the order in which R and S tuples are processed. As a result, stream joins that arbitrarily interleave R and S tuples do not guarantee deterministic processing and require additional synchronization when used by applications in which non-determinism could cause money loss or violation of service-level agreements.

Definition 1: [4] A stream join implementation is *deterministic* if, given the same sequences of input tuples, the same sequence of output tuples will be produced, independently of the tuples' inter-arrival time and processing order.

Example violating determinism Consider tuple t_3 to be delivered and processed after tuple t_5 (e.g., due to a network delay) in the execution presented in Figure 1. In this case, the premature purging of tuple t_2 from W_S would result in a missed comparison (i.e., in a possible output). As shown in [9], this shortcoming is exacerbated when tuples are delivered by multiple R and S physical streams.

In existing work [9], [2], deterministic processing of the three-step procedure is achieved by merging the timestamp-sorted tuples coming from different streams and feeding the join with a timestamp-sorted stream of *ready* [4] tuples.

Definition 2: Let t_i^j be the i-th tuple from timestamp-sorted stream j. t_i^j is *ready* to be processed if $t_i^j.ts < merge_{ts}$, where $merge_{ts} = min_j\{max_i(t_i^j.ts)\}$ is the minimum among the latest timestamps from each timestamp-sorted stream j.

Determinism and processing latency In the literature, such merging is not integrated in the operator itself but rather executed by dedicated operators such as *Input Mergers* [9] or *SUnions* [2]. These dedicated operators are *sequential* and *single-threaded* implementations of the merging procedure and, as a result, represent a potential bottleneck and go against one of our motivations, namely disjoint-parallelism. Moreover, this two-phase procedure (first merge, then process) introduces overheads in the overall processing latency (e.g., due to scheduling of multiple stream operators). As we will show, deterministic processing can be indeed guaranteed without reducing the parallelism degree during these two phases, by focusing on the underlying data structures used in the parallel stream join algorithmic implementation.

Requirements for deterministic stream joins If a tuple is *ready*, no other tuple with a lower timestamp will be delivered by any other stream. Hence, consuming *ready* tuples in timestamp order ensures (i) that no tuple t is removed from window W_R or W_S before all the comparisons involving t are run and (ii) that output tuples are outputted in timestamp order (as presented in Section II, an output tuple's timestamp is set as the highest of the two matching tuples). That is:

Proposition 1: The processing of a sequential stream join by means of the three-step procedure is deterministic if *ready* tuples from R and S are processed in timestamp order.

It should be noted that the three-step procedure does not compare tuples t_R and t_S only if $|t_R.ts - t_S.ts| \leq \texttt{WS}$ (in Figure 1, tuples t_1 and t_4 are compared even if their time distance if 5 while $\texttt{WS}$ is 4). As we prove in Section V-E, this does not affect deterministic processing. The condition $|t_R.ts - t_S.ts| \leq \texttt{WS}$ can be enforced by a modified three-step procedure that, upon reception of t, removes tuples from t's *opposite* window, runs the comparisons and finally adds t to its respective window. We do not focus on such modified procedure. However, it is trivial to extend our findings to it.

Similarly as in [9], we see that a parallel stream join implementation remains deterministic if its processing is equivalent to that of a sequential one (as in Proposition 1).

Proposition 2: Given a deterministic sequential stream join J_S and a parallel stream join J_P sharing P and $\texttt{WS}$, J_P's processing is equivalent (and thus deterministic) to that of J_S if, by processing the same R and S tuples, J_P (1) runs the same set of comparisons run by J_S and (2) produces the same timestamp-sorted stream of output tuples.

It should be noted that Proposition 2 does not require J_S and J_P to share the same number of physical input streams. Hence, implementations that, as *ScaleJoin*, fulfill the proposition's requirements result in deterministic processing also when the tuples fed to J_S by exactly one R and S physical streams are fed to J_P by multiple R and S physical streams.

IV. ScaleJoin

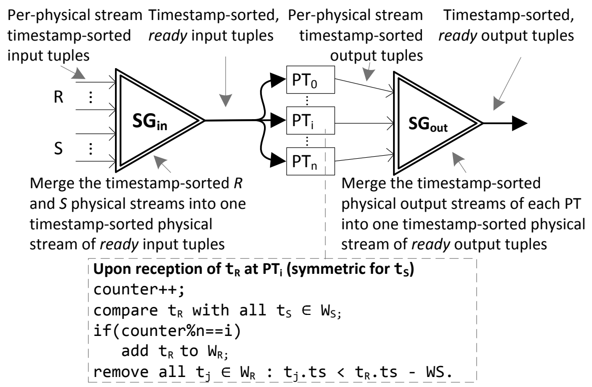
This section overviews *ScaleJoin*'s architecture, presents a sample execution and shows how it addresses the challenges described in the previous sections. We first introduce *ScaleGate*, the abstract data type which allows for the parallelization and balancing of the work.

A. The ScaleGate abstract data type

ScaleGate allows for an arbitrary number of timestamp-sorted streams (i.e., the physical R and S streams), each delivered by one *source* entity (i.e. typically a thread)[2], to be merged into a timestamp-sorted stream of *ready* tuples (cf. Definition 2). At the same time, it allows for an arbitrary number of *reader* entities to consume all the *ready* tuples of the latter stream. *ScaleGate* encapsulates the necessary communication between the source and reader entities in order to decide whether a tuple is *ready* or not. The interface of *ScaleGate* provides the methods:

- `addTuple(tuple,sourceID)`: which allows a `tuple` from the source entity `sourceID` to be merged by *ScaleGate* in the resulting timestamp-sorted stream of *ready* tuples.

[2]This can be extended to allow a physical stream to be delivered by more than one source entity by splitting it in two or more physical streams


Per-physical stream timestamp-sorted input tuples
Timestamp-sorted, *ready* input tuples
Per-physical stream timestamp-sorted output tuples
Timestamp-sorted, *ready* output tuples
R
S
SG_in
PT_0
PT_i
PT_n
SG_out
Merge the timestamp-sorted *R* and *S* physical streams into one timestamp-sorted physical stream of *ready* input tuples
Merge the timestamp-sorted physical output streams of each PT into one timestamp-sorted physical stream of *ready* output tuples

Upon reception of t_R at PT_i (symmetric for t_S)
```
counter++;
compare t_R with all t_S ∈ W_S;
if(counter%n==i)
   add t_R to W_R;
remove all t_j ∈ W_R : t_j.ts < t_R.ts - WS.
```

Figure 2: Overview of *ScaleJoin*'s architecture and parallelization approach.

- `getNextReadyTuple(readerID)`: which provides to the calling reader entity `readerID` the next earliest *ready* tuple that has not been yet consumed by the former.

B. ScaleJoin architecture overview

ScaleJoin allows for the parallel execution of an arbitrary number n of Processing Threads (PTs), each consuming and matching the input tuples delivered by R and S streams. Its processing consists of three stages: (1) delivery of input tuples to PTs, (2) matching of tuples at PTs and (3) collection of PTs' output tuples, as shown in Figure 2.

Delivery of input tuples to PTs We employ a first *ScaleGate* (SG_{in}) to merge the R and S tuples delivered by an arbitrary number of physical R and S input streams (each acting as one source entity) into a single timestamp-sorted stream of *ready* R and S tuples. The different source entities invoke the `addTuple` operation of the SG_{in}.

Matching of input tuples at PTs The timestamp-sorted stream of *ready* R and S tuples is consumed by the n PTs. Each PT acts as a reader entity for SG_{in}. To guarentee deterministic processing, we want each comparison to be run by exactly one PT. At the same time, we want each PT to run a fair share of the overall comparisons (approximately $\frac{1}{n}$) to keep the work balanced. To achieve these goals, we slightly modify the original three-step procedure (as presented in Figure 2) so that R and S tuples are stored in PT's windows in a round robin fashion. Each PT_i maintains a counter of the *ready* tuples being processed and stores a new input *ready* tuple only if $counter\%n$ equals i.

Collection of PTs' output tuples By processing R and S *ready* tuples in timestamp order, each PT delivers output tuples in timestamp order. We rely on a second *ScaleGate* (SG_{out}) to merge the output tuples produced by each PT into a single timestamp-sorted stream of *ready* output tuples. In this case, each PT will act as a source entity of the SG_{out}. The `getNextReadyTuple` method will be invoked on the SG_{out} by the execution unit (e.g. thread) in charge of forwarding the stream join's results.

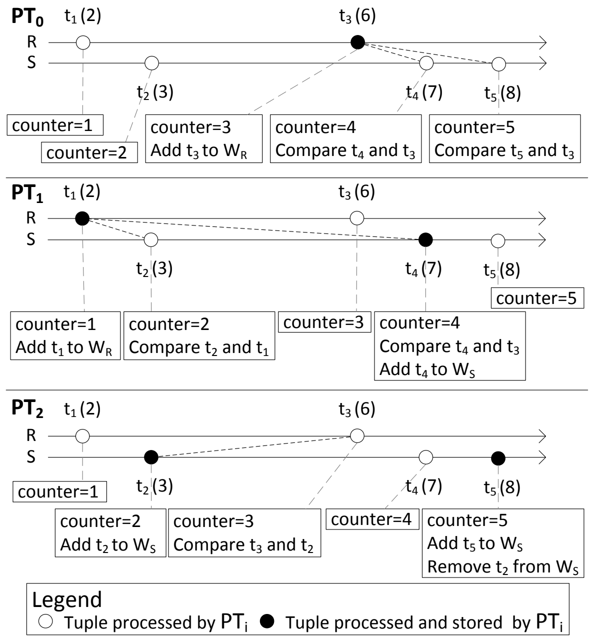

PT0
t1 (2)
t3 (6)
R
S
t2 (3)
t4 (7)
t5 (8)
counter=1
counter=2
counter=3 Add t3 to WR
counter=4 Compare t4 and t3
counter=5 Compare t5 and t3
PT1
counter=1 Add t1 to WR
counter=2 Compare t2 and t1
counter=3
counter=4 Compare t4 and t3 Add t4 to WS
counter=5
PT2
counter=1
counter=2 Add t2 to WS
counter=3 Compare t3 and t2
counter=4
counter=5 Add t5 to WS Remove t2 from WS
Legend
○ Tuple processed by PTi
● Tuple processed and stored by PTi

Figure 3: Sample execution of a *ScaleJoin* instance running with three PTs for the input tuples presented in Figure 1.

Sample execution of ScaleJoin Figure 3 shows how the input tuples presented in Figure 1 would be processed by a *ScaleJoin* instance running 3 PTs. A white circle represents a tuple processed (but not stored) by a PT, while a black circle represents a tuple processed and stored by a PT. Each tuple is processed by all PTs but only stored by exactly one of them in a round-robin fashion. The overall number of comparisons run by *ScaleJoin* is the same as the one run by its centralized counterpart (i.e., they result in the same output tuples, which will be merged by the SG_{out}), while the comparisons are evenly distributed among the PTs.

C. Properties of the proposed methods

We outline below why *ScaleJoin* meets its motivating challenges in an intuitive fashion. Formal statements, proofs and evaluation are given in Sections V and VI.

Deterministic-processing Given Proposition 2, *ScaleJoin* enforces deterministic processing since each comparison run by a sequential stream join is run by exactly one of the PTs (the one that stored the earliest tuple being compared) and output tuples are delivered in timestamp-sorted order.

Disjoint-parallelism *ScaleJoin* does not define any centralized sequential component. Each of the n PTs runs $\frac{1}{n}$ of the overall comparison in a disjoint-parallel fashion while invoking the methods provided by *ScaleGate* to get *ready* input tuples and add output ones concurrently.

Skew-Resilience Thanks to SG_{in}, each new *ready* tuple can be processed by each PT independently of (1) the number of physical streams delivering R and S tuples, (2) variations in the rate with which R and S tuples are delivered and (3) the distribution of R's and S's tuples' values.

Algorithm 1: PTs implementation

```
1  ScaleGate $SG_{in}$, $SG_{out}$; // Input & output ScaleGates,
2                                   // shared among all $PT$s
3  List $W_R$, $W_S$;               // $R$ and $S$ windows
4  int id, n, counter, WS;          // $PT$'s id, # of $PT$s, tuple
5                                   // counter and window size
6  $P$ pred;                        // Predicate
7
8  run()
9    readyTuple = $SG_{in}$.getNextReadyTuple(id)
10   if(readyTuple$\neq$null)
11     counter++;
12     if (isFromR(readyTuple))
13       thisWin = $W_R$; otherWin = $W_S$;
14     else
15       thisWin = $W_S$; otherWin = $W_R$;
16     for (Tuple t $\in$ otherWin)
17       if (pred.holds(readyTuple,t))
18         $SG_{out}$.addTuple(combine(readyTuple,t),id);
19     if (counter%n==id)
20       thisWin.add(readyTuple);
21     for (Tuple t $\in$ thisWin)
22       if (t<readyTuple.ts-$WS$)
23         thisWin.remove(t);
```

Extensibility By relying on the presented three-step procedure, *ScaleJoin* eases the integration of optimization techniques (e.g., equi-joins or band-joins) that, using appropriate data structures at the PTs, minimize the comparisons run by *ScaleJoin* (e.g., hash tables for equi-joins).

V. Algorithmic implementations

We first present the PTs algorithm and discuss how they access SG_{in} and SG_{out}. Subsequently, we present the design and algorithmic implementation of *ScaleGate* Finally, we show that the *ScaleGate* algorithmic implementation is correct and that *ScaleJoin* enforces determinism.

A. Processing Thread

As explained in Section IV, each PT executes a modified version of the sequential three-step procedure. Algorithm 1 presents the steps performed by each PT in detail.

Each PT maintains a reference to the SG_{in} and SG_{out} data structures, W_R and W_S windows, its unique `id`, the total number of PTs `n`, a tuple `counter`, the window size `WS` and the predicate P (L1-6). Each PT starts its execution by retrieving the next available *ready* tuple from SG_{in} (L9). If no *ready* tuple is available (i.e., if the value returned by SG_{in} is null) the PT keeps trying until one is available (L10). Subsequently, the PT proceeds setting the references for `thisWin` and `otherWin` (L12) using the auxiliary method `isFromR` (not shown in the algorithm). The *ready* tuple is then compared with each tuple in the opposite window (`otherWin`). If P holds, an output tuple is added to SG_{out} (L16) using the auxiliary method `combine` (not shown in the algorithm) which creates the output tuple as described in Section II. Subsequently, the *ready* tuple is added to its corresponding window (`thisWin`) (L19). Notice that the condition `counter%n==id` holds for exactly one PT (that is, each tuple is stored in its respective window by exactly one PT). Finally, stale tuples are removed from the *ready* tuple's respective window (`thisWin`) (L21-23).

B. Motivation of ScaleGate's implementation

As discussed in Section IV, *ScaleGate*'s goal is to merge, in a parallel and concurrent fashion, arbitrary numbers of physical streams (e.g., the physical input streams for SG_{in} and the PTs output streams for SG_{out}).

Out of many synchronization choices for the implementation of *ScaleGate*, lock-free (a.k.a. non-blocking) implementations ensure system-wide progress, by guaranteeing at least one of the threads operating on the data structure to make progress independently of the behavior of other threads. Such implementations demonstrate higher scalability and better fairness when compared with other coarse- or fine-grain locking mechanisms [3] and hold across different hardware architectures. Motivated by the above we target for a lock-free concurrent implementation of *ScaleGate*.

A basic requirement for an implementation of *ScaleGate* is to maintain items in a sorted manner. Tree-like implementations are not efficient in concurrent environments due to the strong dependencies in balancing operations [11]. On the contrary, shared concurrent skip lists [17], [11] are used extensively. In a nutshell, skip lists maintain a sorted linked list of elements (e.g., tuples), while allowing for concurrent insertions and deletions with overhead that is probabilistically logarithmic. This is made possible by multiple levels (pointers) for each element, acting as shortcuts for quickly locating the position of an element. The number of additional levels for each element is chosen randomly.

Inspired by skip lists, which themselves do not provide support for determining *ready* tuples or other similar synchronization, we design a multi-level pointer mechanism adapted to the *ScaleGate* requirements. Such adaption enables fine-grained synchronization that boosts parallelism and is carried out (1) by making *ScaleGate* inherently aware of the concept of *ready* tuples and (2) by exploiting the specific access patterns of *ready* tuples (e.g., consumed in timestamp order from SG_{in}, L9) and thus allowing for a more lightweight implementation than the general purpose delete operations of skip lists.

C. ScaleGate algorithmic implementation

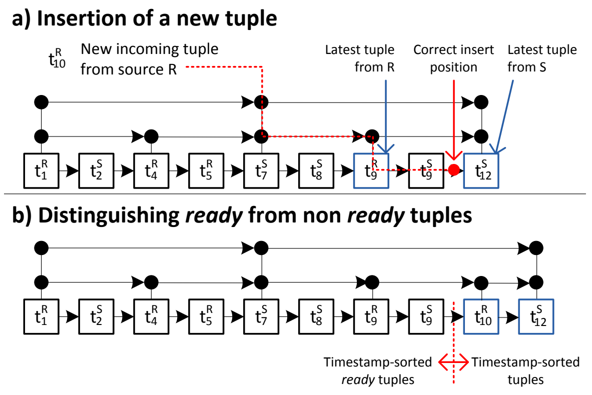
Algorithm 2 presents the *ScaleGate* implementation, in Java-like pseudocode. The *ScaleGate* consists of nodes, each containing a `Tuple`, its source id and an array of references, `next`, for the multi-level connections. The `tail` is statically allocated and indicates the end of the list.

The `addTuple` operation inserts a `tuple` in the appropriate position in the list according to its timestamp. The `update` array (i.e. thread local) keeps references to the nodes closer in each level to the latest inserted node. As the tuples from each source arrive in increasing timestamp order,

Algorithm 2: *ScaleGate* implementation

```
25 Node head, update[maxlevels] // Thread local variables
26                              // maxlevels is a
27                              // constant parameter
28 Tuple[#sourceIDs] written // Shared array of the
29                           // last written tuples
30 Node tail // Shared variable, pointing to a dummy
31           // statically allocated node
32
33 def Node
34   Node next[maxlevels]
35   Tuple tuple
36   int sourceID
37
38 getNextReadyTuple(readerID)
39   nextNode = head.next$_0$
40   if(nextNode$\neq$tail
41     $\wedge$written$_{nextNode.sourceID}$ $\neq$nextNode.tuple)
42     head = nextNode
43     return nextNode.tuple
44   return null
45
46 addTuple(tuple,sourceID)
47   levels = getLevelHeight()  // get random height
48                              // up to maxlevels
49   newnode = new Node(tuple, sourceID)
50   curnode = update$_{maxlevels-1}$
51   for(i=maxlevels-1 downto 0)
52     next = curnode.next$_i$
53     while(next $\neq$ tail $\wedge$ next.ts<tuple.ts)
54       curnode = next
55       next = curnode.next$_i$
56       update$_i$ = curnode
57   for(i=0 to levels)
58     levelinsert(update$_i$, newnode, tuple.ts, i)
59   written$_{sourceID}$ = newnode.tuple
60
61 levelinsert(fromnode, newnode, ts, level)
62   while(true)
63     next = fromnode.next$_{level}$
64     if(next==tail $\vee$ next.ts>ts)
65       newnode.next$_{level}$ = next
66       if(CAS(fromnode.next$_{level}$, next, newnode)) break
67     else fromNode = next
```

the search for the correct position is optimized by starting from the highest level node closer to the latest inserted tuple from the same source (L50), instead of starting from the beginning of the list. The rest of the `update` array is used to temporarily store references to the levels that need to be connected with the new node. In detail, during an `addTuple` operation, the number of levels that the new node will hold is decided with the `getLevelHeight` method (L47), according to the standard skip list [17]. Afterwards, the shortcuts are traversed in order for the appropriate position of the new node to be found (L51-56). The node is then inserted on each level it should be part of, with the use of the `levelinsert` method (L58). This helper method checks if the node stored in the update array is still the prior node. If not, it traverses the list until it finds the right node. The next field of the prior node is then changed to point to the new one, using the atomic compare and swap (CAS) operation. If it fails, it means another node was inserted at the same time by another source. In this case we need to search for the prior node again and repeat. Once



a) Insertion of a new tuple
t^R_{10} New incoming tuple from source R
Latest tuple from R
Correct insert position
Latest tuple from S
t^R_1 t^S_2 t^R_4 t^R_5 t^S_7 t^S_8 t^R_9 t^S_9 t^S_{12}
b) Distinguishing *ready* from non *ready* tuples
t^R_1 t^S_2 t^R_4 t^R_5 t^S_7 t^S_8 t^R_9 t^S_9 t^R_{10} t^S_{12}
Timestamp-sorted *ready* tuples
Timestamp-sorted tuples

Figure 4: Insertion of a new tuple in the SG_{in} *ScaleGate* and differentiation between *ready* and non *ready* tuples.

the node has been inserted, a reference to it is kept in the `written` array (L59), indexed by the `sourceID`, so that the tuple is not read (during the `getNextReadyTuple` operation) until a newer one is received from the same `sourceID`. Note that while the `written` array is shared among source entities (threads), each of its elements is exclusively updated by a specific sourceID thread.

The `getNextReadyTuple` method ensures that the calling `readerID` gets all the *ready* tuples in timestamp order. For each calling `reader` a local view of the `head` is kept, and the lowest level of the list is traversed. If the current node is referenced by the respective cell of the `written` array[3] (i.e. the tuple is not *ready* yet) `null` will be returned. Reading a `written` array cell by the reader, assumes an implementation language with a well defined memory model (e.g., C++11 or Java), or appropriate barriers.

Nodes are freed when they are no longer accessible from the nodes that are referred to by the local views of the `head` and `update`$_{maxlevels-1}$. Thus, the prefix of nodes in the list, up to the first node that is referenced by a local `head` or an entry of the `update` array, can be easily garbage collected at any point. In unmanaged environments, lock-free reference counting techniques can be used for managing the nodes [8]. In both cases, resilience to the classical ABA problem is guaranteed [14]. Finally, note that if the CAS instruction is not available in the underlying architecture, the LL/SC primitive is a common equivalent alternative [11].

Example Figure 4 shows how R and S tuples (in the example, each delivered by 1 physical stream) are maintained and inserted in SG_{in} (superscript and subscript of each tuple refer to its stream and timestamp, respectively).

D. *ScaleGate correctness*

In this section we argue about the liveness and safety properties, namely *lock-freedom* [11] and *linearizabil-*

[3]This check is valid as `written`$_{sourceID}$ is never null, and is updated with nodes that have monotonically increasing timestamps. If an old value of `written`$_{sourceID}$ is observed, it will result to an unsuccessful call of the `getNextReadyTuple`, forcing the caller to retry (L10).

ity [12], of the *ScaleGate* implementation. The former ensures that at least one of the threads performing operations on the data structure will make progress in a bounded number of its own steps. According to the definition of linearizability [12], [11], every method call should appear to take effect at some point (linearization point) between its invocation and response. Thus, given a history of concurrent operations and by using the linearization points, we are able to define a total order in the execution, which is consistent with the real-time ordering of the operations and with the sequential semantics of the data structure.

Theorem 1: The *ScaleGate* implementation presented in algorithm 2 is lock-free and linearizable.

Proof: Method `getNextReadyTuple` returns in a bounded number of its own steps. The `addTuple` method call will fail to return only if the CAS instruction on L66 fails (i.e., if another concurrent call of `addTuple` from another thread makes progress). Thus the *ScaleGate* implementation is lock-free.

Concurrent calls of `addTuple` appear to each-other to take effect only after a successful execution of the CAS instruction on L66, which is the linearization point among such calls. The linearization point of `getNextReadyTuple` is the read of the `written` array entry for the respective `sourceID` during the check on L41, where the condition for a *ready* tuple is checked. That is, in the case of concurrent calls of `getNextReadyTuple` and `addTuple`, the linearization point of the latter is the update of the `written` array on L59. Thus there is a linearization point for all the method calls of the *ScaleGate* implementation. ■

E. Proof of deterministic processing

According to Proposition 2, we show that *ScaleJoin* enforces deterministic processing, based on the following.

Lemma 1: A tuple $t_R \in R$ (respectively $t_S \in S$) is only stored by one PT in the corresponding window W_R (respectively W_S).

Lemma 2: A *ready* tuple $t_R \in R$ (respectively $t_S \in S$) is consumed from SG_{in} by all PTs.

Theorem 2: ScaleJoin implementation enforces deterministic processing, equivalent to that of a deterministic sequential stream join.

Proof: Without loss of generality, let $t_R \in R$, $t_S \in S | t_R.ts > t_S.ts$ be a pair of tuples compared by a sequential stream join. The same comparison is run by one and only one PT in *ScaleJoin*, the one that will have previously stored t_S in its corresponding window W_S (Lemma 1), which will further have consumed the *ready* tuple t_R (Lemma 2).

Since all PTs process a timestamp-sorted stream of *ready* tuples, they will produce timestamp-sorted streams of output tuples that are merged by SG_{out}, thus resulting in the same sequence of output tuples produced by the sequential counterpart. ■

VI. Evaluation

This section presents *ScaleJoin*'s performance results. We first introduce the benchmark used in the evaluation. Subsequently, we study (i) *ScaleJoin*'s scalability in terms of both *comparisons/second (c/s)* and *tuples/second (t/s)* and (ii) the rate at which tuples can be added to and retrieved from a *ScaleGate* instance. We continue by measuring *ScaleJoin*'s *end-to-end processing latency* (or simply *latency* in the remainder). Finally, we study how well *ScaleJoin* addresses the skew-resilience challenge by measuring how its processing load is distributed among the processing threads PTs for different numbers of physical input streams and different rate behaviors. We conclude summarizing the results.

We also set side-by-side (in terms of throughput and latency) our Java-based *ScaleJoin* with the C-based Handshake join [18], [16]. We do this to position *ScaleJoin*'s performance with respect to the shared-memory parallel stream join mentioned as the fastest in the literature. We do not evaluate Handshake's skew-resilience as it does not allow to process more than one physical R or S input streams.

Evaluation setup We follow the common benchmark used to evaluate CellJoin [6] and Handshake joins [18], [16]. R tuples are composed by attributes $\langle ts,x,y,z \rangle$, where x,y,z are of types `int`, `float` and `char[20]`, respectively. S tuples are composed by attributes $\langle ts,a,b,c,d \rangle$, where a,b,c,d are of types `int`, `float`, `double` and `bool`, respectively. An output tuple $\langle ts,x,y,z,a,b,c,d \rangle$ is created for each pair of tuples t_R,t_S such that:

$$t_R.x \geq t_S.a - 10 \texttt{ AND } t_R.x \leq t_S.a + 10 \texttt{ AND } t_R.y \geq t_S.b - 10 \texttt{ AND } t_R.y \leq t_S.b + 10$$

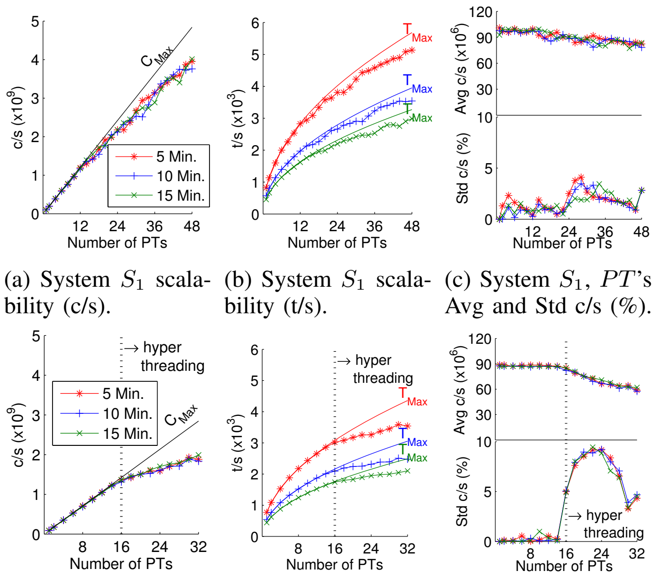
Values for attributes x,y,a,b are drawn from a uniform distribution in the interval $[1-10{,}000]$. 1 out of each 250,000 comparisons results in an output tuple, on average [16].

ScaleJoin is evaluated using two different systems: (i) *System* S_1, equipped with a 2.6 GHz AMD Opteron 6230 (48 cores over 4 sockets), implementing a non-uniform memory access (NUMA) architecture, and 64 GB of memory; and (ii) *System* S_2, equipped with a 2.0 GHz Intel Xeon E5-2650 (16 cores over 2 sockets) and 64 GB of memory. This setup allows us to study *ScaleJoin*'s scalability across different architectures and when using hyper-threading (system S_2). S_1 and S_2, with different numbers of sockets, further enhance NUMA effects when accessing shared memory.

Experiments start with a warm-up phase; measurements are taken during the steady-state phase, averaging each value over 5 repetitions. Dedicated threads inject input tuples and collect output tuples.

Scalability evaluation of ScaleJoin

Similarly to CellJoin and Handshake joins, we first assess the scalability of *ScaleJoin* for one physical R and S streams with equal input rates, different window sizes and



(a) System S_1 scalability (c/s). (b) System S_1 scalability (t/s). (c) System S_1, PT's Avg and Std c/s (%).

(d) System S_2 scalability (c/s). (e) System S_2 scalability (t/s). (f) System S_2, PT's Avg and Std c/s (%).

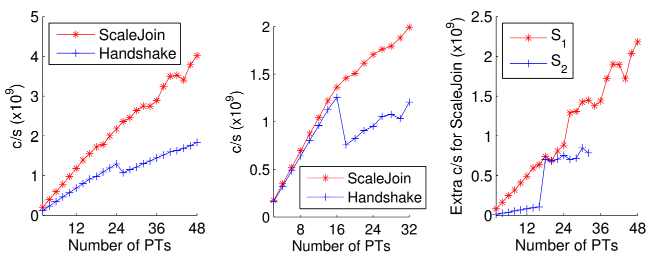
Figure 5: Scalability evaluation (c/s and t/s) for S_1 and S_2, different window sizes and increasing number of PTs.

an increasing number of PTs. For window sizes of 5, 10 and 15 minutes, we measure the maximum number of c/s and t/s sustained by *ScaleJoin*. Moreover, to highlight the balanced work, we also measure the average number of c/s (*Avg c/s*) run by each PT and the corresponding standard deviation (*Std c/s*) in percentage.

Bounds of expected results The number of c/s for a given window size of WS seconds and input rate of T t/s (same for R and S streams) is $2 \times WS \times T^2$ (Section II). Being C_{max} the maximum number of c/s executed by one PT, the expected maximum number T_{max} of t/s processed by such PT is $T_{max} = \sqrt{\frac{C_{max}}{2 \times WS}}$ (that is, T_{max} depends on WS). A perfectly linear scalability when moving from 1 to n PTs would result in $n \times C_{max}$ c/s and $\sqrt{n} \times T_{max}$ t/s. I.e., in the *ideal case*, *ScaleJoin* is expected to scale linearly on the number of PTs in terms of c/s and to scale proportionally to the square root of the number of PTs in terms of t/s.

System S_1 scalability Results for system S_1 are shown in Figures 5a, 5b and 5c (solid lines represent the maximum achievable scalability C_{max} and T_{max}). As expected, the maximum number of c/s grows linearly, while the maximum number of t/s grows as the square root of the increasing number of PTs. *ScaleJoin* can achieve approximately 4 billion c/s, resulting in rates of approximately 5,100 t/s, 3,500 t/s and 3,000 t/s for window sizes of 5, 10 and 15 minutes, respectively. While the Avg c/s changes from approximately 100 to 80 millions per PT, when changing from 1 to 48 PTs, the overall workload is evenly distributed, as shown by the Std c/s that does not exceed 4%.

System S_2 scalability Results for system S_2 are shown



(a) c/s for system S_1 (b) c/s for system S_2 (c) Difference in c/s.

Figure 6: Throughput in *ScaleJoin* and Handshake joins.

in Figures 5d, 5e and 5f. When using less than 16 PTs (the number of available physical cores), *ScaleJoin* achieves an almost perfectly linear scalability while having very balanced work among all the PTs (the standard deviation up to 16 PTs does not exceed 2%). *ScaleJoin* can achieve a rate of approximately 1.4 billion c/s, resulting in rates of approximately 3,000 t/s, 2,100 t/s and 1,750 t/s for window sizes of 5, 10 and 15 minutes, respectively. When using hyper-threading, despite a general throughput degradation, *ScaleJoin* is still able to achieve a linear scalability (with a milder slope) and an extra 500 millions c/s while having a balanced work (whose Std c/s does not exceed 10%).

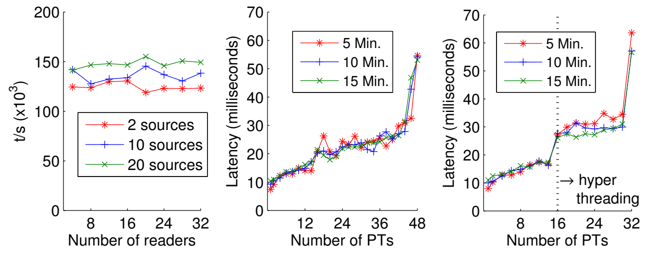
Relation with Handshake join Results for *ScaleJoin* and Handshake join are presented in Figures 6a,6b,6c. For system S_1, the *difference* between the maximum number of c/s sustained by *ScaleJoin* and the Handshake join increases linearly with the number of processing threads. For 48 cores, *ScaleJoin* performs approximately 2.5 billion c/s more than Handshake join. For system S_2, a significant step up is observed when the number of processing threads exceeds the available cores (hyper-threading). In this case, *ScaleJoin* achieves almost 1 billion c/s more than Handshake join.

Performance of *ScaleGate* In order to show that the highest achieved throughput is not limited by *ScaleGate*, we report in Figure 7a the rate at which tuples can be added to and retrieved from a *ScaleGate* instance for increasing numbers of source and reader entities (for System S_2). The rate increases with the number of source entities and does not degrade for an increasing number of reader entities (even when using hyper-threading with more than 16 source entities). Moreover, the rate grows to 150,000 t/s, approximately, 50 times higher than the highest processing throughput observed (3,000 t/s).

Latency evaluation of ScaleJoin

As discussed in [1], low latency is essential for stream joins used in time-sensitive applications such as option pricing (tolerating latencies of few seconds, at maximum). We measure latency at the highest throughput achieved for each number of PTs (that is, for the experiments in Figure 5).

As presented in Figures 7b and 7c, *ScaleJoin* achieves a



(a) *ScaleGate* performance (System S_2). (b) *ScaleJoin* latency for system S_1. (c) *ScaleJoin* latency for system S_2.

Figure 7: *ScaleGate* throughput for increasing numbers of source and reader entities (7a) and latency evaluation of *ScaleJoin* for different window sizes (at the max throughput) and for an increasing number of PTs (7b,7c)

very low latency, which does not exceed 70 ms. Such latency increases linearly with the increasing number of PTs. This is because the increasing number of PTs results in a lower per-PT output rate. Given the definition of *ready* tuple (cf. Definition 2), this results in a longer time (i.e., a higher latency) for each output tuple to become *ready* at SG_{out}. A "jump" of approximately 10 ms is observed when using 16 (the number of available cores) or more PTs in system S_2.

Relation with Handshake join As shown in [16], the original Handshake join latency is, by design, half of the window size (i.e., a window size of 15 minutes can result in latencies up to 7.5 minutes). The improved low-latency Handshake join, similarly to *ScaleJoin*, achieves *processing* latencies in the realm of milliseconds. Nevertheless, in order to ensure deterministic sorting of output tuples, it relies on punctuation tuples and external buffers maintaining the output tuples to be sorted. The authors do not specify the latency introduced by the sorting itself, but specify that such buffers can grow in size up to 30 thousands tuples, approximately. Based on the highest throughput we observe (i.e., 4 billion c/s) and on the fact that 1 out of 250,000 comparisons results in an output tuple, on average, more than 2 seconds are needed to fill such a buffer. Hence, the sorting of output tuples would result in latencies in the realm of seconds. In this sense, *ScaleJoin* is able to provide orders of magnitude lower latency while guaranteeing deterministic processing.

Skew-resilience evaluation

We evaluate here *ScaleJoin*'s capability of maintaining PTs' work balanced when tuples are delivered by multiple rate-varying, bursty physical streams. We introduce 3 different case-studies in which tuples are delivered by distinct numbers of physical streams. Results refer to system S_2.

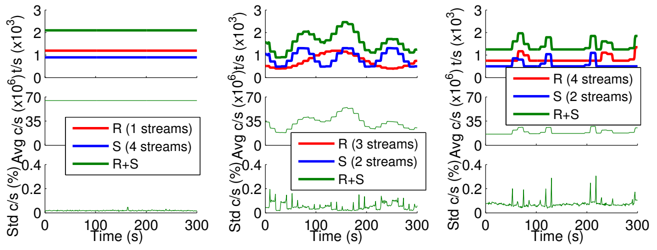
Constant distinct rates This case-study shows how *ScaleJoin* is able to achieve a perfectly balanced work among the PTs when R and S tuples are delivered by multiple physical streams at different (but yet constant) rates. In the experiment, 1 R and 4 S physical streams deliver 1,200 and 900 t/s, respectively. Comparisons are executed by 10 PTs. Figure 8a presents the experiment results (for a period of 5 minutes). The upper part of the figure presents the input rates at which R and S tuples are delivered. The middle part of the figure presents the Avg c/s run by each of the 10 PTs. Finally, the lower part presents the resulting Std c/s (in percentage). As it can be observed, approximately 65 million c/s are run by each of the PTs, perfectly balanced as evidenced by the Std c/s stable around 0.05%.

Fluctuating distinct rates In this case-study, 3 R and 2 S physical streams deliver tuples with rates that oscillate following a sinusoidal function with different amplitude and period. Comparisons are executed by 12 PTs. As shown in Figure 8b, the number of c/s performed by each PT fluctuates accordingly to the input rates, from a minimum of 18.5 to a maximum of 55 million c/s, remaining perfectly balanced, as evidenced by the Std c/s stable around 0.1%.

Constant distinct rates with peaks In this third case-study, we evaluate how balanced is the work among the PTs when sudden peaks appear in the input streams. R and S tuples are delivered by 4 and 2 physical streams while the overall comparisons are run by 14 PTs. Also in this case, as shown in Figure 8c the overall work is perfectly balanced, with a Std c/s stable around 0.1%.

Highlights of the experimental evaluation study

ScaleJoin is able to meet the bounds of the expected throughput across different NUMA architectures and to be skew-resilient (keeping a perfectly balanced work among PTs independently of the number of physical input streams and their rate fluctuations). Similar performance behaviour is observed in the two systems S_1 and S_2; finer-grain differences are due to the different hardware architectures (e.g. HyperThreading in S_2), frequencies and access latencies to the different distributed memory banks (NUMA regions).



(a) Distinct constant rates, 10 PTs. (b) Distinct fluctuating rates, 12 PTs. (c) Distinct rates with peaks, 14 PTs.

Figure 8: Skew-resilience evaluation (system S_2). Average number of c/s run by each PT and standard deviation (in percentage) for distinct numbers of physical streams.

VII. Related work

Both shared-nothing and shared-memory parallelization techniques exist in the literature for stream joins.

Shared-nothing techniques such as [9], [1] allow for parallel analysis to span multiple distinct nodes. Their scope differs from *ScaleJoin*'s. StreamCloud [9] focuses on the parallelization of generic stream operators. As a consequence, it results in lower processing throughput than join-specific parallelization techniques (e.g., because of tuple duplication for the parallelization of stream joins). Photon [1] focuses on geographically distributed systems. Elseidy et al [5] present an adaptive operator for shared-nothing parallel joins, but in a data flow setting that does not consider sliding windows.

Shared-memory techniques can be further differentiated between architecture-specific [6] and more hardware-independent and generic ones [18], [16]. These approaches address only partially the motivating challenges behind *ScaleJoin*. A common assumption is for tuples of a logical stream to be delivered by a single physical stream [6], [18], [16]. Hence, differently from *ScaleJoin*, they cannot be leveraged when multiple physical streams are fed to a parallel stream join. Moreover, they do not discuss or prove to enforce deterministic processing [6], [18], [16] (the closest discussion focuses on the deterministically sorted output streams provided by the low-latency Handshake join [16]). Finally, they rely on central coordinated partitioning and replicating techniques and thus do not provide disjoint-parallelism [6] or do not evaluate their performance in the presence of fluctuating, bursty streams.

VIII. Discussion and Future Work

In this work we propose *ScaleJoin*, a scalable disjoint-parallel, shared-memory stream join that provides deterministic high-throughput and low-latency joining of tuples delivered by arbitrary numbers of streams. These properties are crucial to leverage shared-memory parallel stream joins in demanding streaming applications. We introduce *ScaleGate*, the data object that provides *ready* tuples and allows for arbitrary numbers of processing threads to run comparisons in a disjoint-parallel fashion. We built *ScaleJoin* relying on *ScaleGate*, enabling high processing throughput and low processing latency while maintaining a balanced work (among the processing threads) when input tuples are delivered by rate-varying and bursty streams.

From a broader perspective, we show that processing does not necessarily imply bottlenecks in the processing of tuples. Suitable shared data objects and lock-free algorithmic implementations allow for efficient, concurrenct and consistent processing and open up the way for new benefits and research paths in parallel streaming applications. Interesting future steps are to extend *ScaleJoin* to include the processing of out-of-order tuples from a given data source, include optimized implementations of equi-joins and band-joins and to study hybrid implementations that leverage both multi-core CPUs and GPUs.

Acknowledgments: The research leading to these results has been partially supported by Chalmers Transportation Area of Advance, by the collaboration framework of Chalmers Energy Area of Advance project "SN09: EXAMINE", by the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183 and by the Swedish Research Council (Vetenskapsrådet) project "Fine-grain synchronization in parallel programming", contract nr. 2010-4801.

References

[1] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 international conference on Management of data*, 2013.
[2] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM TODS*, 2008.
[3] D. Cederman, B. Chatterjee, Nhan Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013.
[4] D. Cederman, V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation . In *Proceedings of the 26th ACM SPAA*, 2014.
[5] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and Adaptive Online Joins. *Proc. VLDB Endow.*, 7(6):441–452, Feb. 2014.
[6] B. Gedik, R. R. Bordawekar, and S. Y. Philip. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal*, 2009.
[7] P. B. Gibbons. Big data: Scale down, scale up, scale out. In *Parallel and Distributed Processing Symposium, 2015. IPDPS 2015. IEEE International*, page 3, May 2015.
[8] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, Aug. 2009.
[9] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE TPDS*, 2012.
[10] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM DEBS*. ACM, 2015.
[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12, 1990.
[13] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings. 19th International Conference on Data Engineering.*, 2003.
[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6), June 2004.
[15] M. M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 2013.
[16] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proceedings of the VLDB Endowment*, 2014.
[17] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 2005.
[18] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.