PPOIJ: Shared-Nothing Parallel Patterns for Efficient Online Interval Joins over Data Streams

Gabriele Mencagli
gabriele.mencagli@unipi.it
University of Pisa
Pisa, Italy

Yuriy Rymarchuk
y.rymarchuk@studenti.unipi.it
University of Pisa
Pisa, Italy

Dalvan Griebler
dalvan.griebler@pucrs.br
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil

Abstract

Joining data streams is a fundamental stateful operator in stream processing. It involves evaluating join pairs of tuples from two streams that meet specific user-defined criteria. This operator is typically time-consuming and often represents the major bottleneck in several real-world continuous queries. This paper focuses on a specific class of join operator, named *online interval join*, where we seek join pairs of tuples that occur within a certain time frame of each other. Our contribution is to propose different parallel patterns for implementing this join operator efficiently in the presence of watermarked data streams and skewed key distributions. The proposed patterns comply with the shared-nothing parallelization paradigm, a popular paradigm adopted by most of the existing Stream Processing Engines. Among the proposed patterns, we introduce one based on hybrid parallelism, which is particularly effective in handling various scenarios in terms of key distribution, number of keys, batching, and parallelism as demonstrated in our experimental analysis.

CCS Concepts: • **Information systems → Stream management**; • **Computing methodologies → Parallel computing methodologies**.

Keywords: Stream Processing, Online Interval Join, Parallel Patterns, Multicores, Parallel Programming

ACM Reference Format:
Gabriele Mencagli, Yuriy Rymarchuk, and Dalvan Griebler. 2025. PPOIJ: Shared-Nothing Parallel Patterns for Efficient Online Interval Joins over Data Streams. In *The 19th ACM International Conference on Distributed and Event-based Systems (DEBS '25), June 10–13, 2025, Gothenburg, Sweden.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3701717.3730542

DEBS '25, June 10–13, 2025, Gothenburg, Sweden

ACM ISBN 979-8-4007-1332-3/2025/06
https://doi.org/10.1145/3701717.3730542

1 Introduction

In the rapidly evolving landscape of data-driven applications, the need for efficient data processing has become critical across industries such as IoT, finance, and cybersecurity. Data stream processing [3] (DSP) is a well-established computing paradigm that enables the continuous processing of data streams. The primary goal of DSP applications, also known as continuous queries, is to extract actionable intelligence, insights, and hidden knowledge from data streams, efficiently and effectively reporting results to end-users. Continuous queries achieve this by transforming input streams through a series of operators connected in data-flow graphs. These operators perform a wide range of transformations, from relational operations on structured streams to general computations on streams with arbitrary data types.

Stream Processing Engines [5] (SPEs) are frameworks designed to support the execution of continuous queries across various computational resources, ranging from single scale-up machines to distributed architectures and cloud environments. To enhance query processing speed, SPEs enable parallel processing by executing different operators simultaneously on different or the same data. Additionally, they replicate each operator into multiple instances, which are run by different threads on separate data items.

Scaling operator performance through parallel processing can be challenging for stateful operators that maintain internal data structures representing the stream history. Since the results of these operators depend on their internal state as well as the current inputs, parallelization techniques must address the challenge of correctly partitioning the state and distributing input items to the operator replicas.

This paper investigates parallel solutions, also known as *parallel patterns*, for the online interval join (OIJ) operator [17]. The OIJ operator has numerous practical applications. For instance, an online shopping platform can recommend products to users based on predefined features by joining data inputs from historical orders within a specific time period. To achieve this, the OIJ operator retains recently received inputs from two input streams. Upon the arrival of a new input, it identifies eligible join pairs by searching for inputs in the opposite stream that fall within a specified time interval relative to the timestamp of the new input.

One of the main challenges in the OIJ parallelization is to provide load balancing among replicas, which might depend on the frequency of distinct keys in the streams. A clever assignment of keys to replicas does not always solve the imbalance. In cases of skewed distributions or a limited number of keys, the processing of inputs with the same key attribute must be parallelized across multiple replicas of the operator. A previous work, SplitJoin [11], distributes all inputs to all replicas, resulting in communication overheads and limited scalability. A recent solution, Scale-OIJ [18], overcomes such a limitation by leveraging a novel sharing technique that allows more replicas to process the same key. However, this approach relies on shared data structures, introducing implicit synchronization across threads.

In this paper, we propose a set of parallel patterns for the OIJ operator. These patterns follow the shared-nothing parallelization paradigm, where replicas do not share data. This approach avoids synchronization and race conditions, addressing limitations of previous works [11, 18]. Our contributions are as follows:

- we first propose a data parallel pattern where inputs are distributed to all replicas, mimicking SplitJoin [11]. However, we eliminate some limitations of SplitJoin such as the use of a single physical input stream and the strict ordering requirement of inputs that must be processed in order in SplitJoin;
- despite enhancements, we critique the data parallel approach for its high communication overheads despite perfect load balancing. We propose a hybrid parallelism pattern where the same key is processed by more replicas (but not necessarily all) without sharing state, unlike prior work [18]. Our heuristic solution dynamically determines the set of replicas for each key, balancing workload with minimal splitting and trading off load balancing with communication overheads;
- we discuss implementation pitfalls for all proposed patterns to avoid missing or duplicate results.

Additionally, we provide an experimental analysis comparing the patterns against standard key-based parallelization and state-of-the-art solutions.

The structure of this paper is as follows. § 2 introduces the background and motivation. § 3 presents the definition of our patterns, while their implementation is discussed in § 4. § 5 presents our experimental evaluation, § 6 reviews related works, and § 7 draws the conclusions of this paper.

2 Background

This section introduces key concepts essential for the upcoming discussion on stream processing, the OIJ operator, and the WindFlow library. The WindFlow library serves as our target DSP framework, where we will examine the performance of the OIJ operator using various parallel patterns as implementation strategies.

2.1 Data Stream Processing

DSP has its origins in the database community years ago [3]. Initially designed to process *structured streams*—unbounded sequences of data records, also known as *tuples*—DSP has received renewed attention in recent years. Continuous queries are directed acyclic data-flow graphs. In these graphs, vertices represent *operators* that perform intermediate data transformation stages, while edges model *data streams* that connect two operators and represent data dependencies.

Operators are *stateless* if they process incoming tuples without keeping any history of the received inputs. Each output tuple is computed solely based on the corresponding input tuple. More complex operators maintain an internal state. These are called *stateful* operators, where the state can be user-defined or built into the underlying SPE. Among stateful operators with a built-in state concept, *windowed operators* [16] play a central role in DSP. They often perform aggregation over tuples within the same window boundary, defined by the number of tuples or timestamps, and partitioned by the key attribute associated by the user to each input tuple.

2.2 Join Operators

Join operators [15, 17] are fundamental in DSP because they enable users to identify correlations and dependencies between two different streams. However, they pose implementation challenges since the original join operators of relational algebra would require maintaining the entire history of past tuples received from each input stream, which is impractical due to computational and memory constraints. For this reason, the join semantics has usually been restricted by considering only the most recent tuples as potential candidates for the join predicate.

Modern SPEs provide two classes of join operators. *Window-based join* applies the join predicate to pairs of tuples (one per stream) that belong to the same corresponding window, e.g., the same time range $[start, end)$. Consecutive windows can be non-overlapping (tumbling) or overlapping (sliding). In contrast, the OIJ operator considers one of the two streams as the *base* stream, while the other is the *probe* stream. The operator emits pairs $\langle t_a, t_b \rangle$, where t_a belongs to stream A (base) while t_b from stream B (probe), provided that: *i)* the timestamp of t_b (denoted by $t_b.ts$) lies within a relative time interval to the timestamp $t_a.ts$; *ii)* the two tuples have a common key attribute, i.e., $t_a.k == t_b.k$; *iii)* an optional joining condition cond over the fields of t_a and t_b evaluates to *true*.

The OIJ operator is configured to use two *time boundaries*, lwr, upr specified by the user. Let $\mathcal{R}(t_a)$ be the set of OIJ results triggered per tuple $t_A \in$ A. This set is defined as:

$$\mathcal{R}(t_a) = \{\langle t_a, t_b \rangle \mid t_a.ts + \texttt{lwr} \leq t_b.ts \leq t_a.ts + \texttt{upr}, \quad t_b \in \texttt{B}, \tag{1}$$

$$t_a.k == t_b.k,$$
$$\text{cond}(t_a, t_b) == true\}$$

The results of the OIJ are defined as follows: $A \bowtie_{oij} B = \bigcup_{\forall t_a \in A} \mathcal{R}(t_a)$. Fig. 1 illustrates an example of OIJ processing with two streams, where tuples are represented by circles. The lower and upper bounds are used to identify a partition of tuples from the probe stream that should be considered in the join processing with each tuple from the base stream.

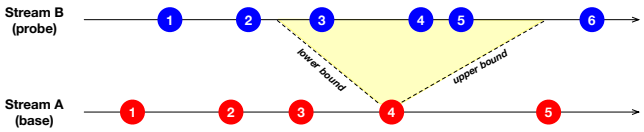

Figure 1. Online interval join (OIJ).

The computational steps of the basic OIJ algorithm can be outlined as follows:

- a new input tuple is received from one of the two streams;
- if the tuple is t_a from the base stream A, the OIJ operator computes the joining condition cond with all previously received tuples t_b from the probe stream B that have the same key attribute and respect the time constraint $t_a.ts + \texttt{lwr} \leq t_b.ts \leq t_a.ts + \texttt{upr}$;
- if the tuple is t_b from the probe stream B, the OIJ operator computes the joining condition cond with all previously received tuples t_a from the base stream A that have the same key attribute and respect the time constraint $t_b.ts - \texttt{upr} \leq t_a.ts \leq t_b.ts - \texttt{lwr}$;
- all pairs of tuples that meet the aforementioned conditions are emitted, and the input tuple is stored in the state buffer of tuples for the base/probe stream, respectively;
- tuples that are no longer useful for joining with future tuples (i.e., they are too old based on their timestamp) are purged from the corresponding state buffer.

The last point poses some challenges based on the assumptions made about the ordering of tuples in the input streams. Further details on this point will be provided at the end of this section.

2.3 Motivation

OIJ is a computationally expensive task of continuous queries. Its parallelization is crucial to achieve adequate levels of throughput and limited latency, as required by various real-world use cases. A critical aspect of parallelization is the choice underpinning the assignment between keys and OIJ replicas. This is illustrated in Fig. 2(left), where we assume two OIJ replicas and three keys, k_0, k_1, k_2, with frequencies of 60% for k_0 and 20% for both k_1 and k_2.

The first case (key-based parallelism) shows a strategy where each key is assigned to exactly one replica, leading to load imbalance. The second strategy (data parallelism) assigns all keys to both replicas, evenly distributing the load but incurring higher overheads due to key splitting (i.e., replication of internal data structures and broadcast distribution of all-tuples-to-all-replicas). The broadcast distribution of all tuples to all replicas is detrimental to performance, as shown in Fig. 2(right) by the normalized throughput of a light computational benchmark with varying broadcast degrees (i.e., the number of recipients a single tuple reaches). The third case (hybrid parallelism) aims for good load balancing with limited key splitting, trading off balancing with runtime overheads. This paper implements and studies these three strategies using the shared-nothing paradigm.

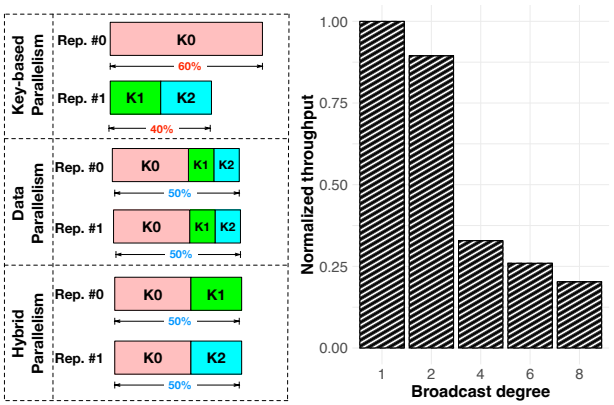

Figure 2. Three parallelization strategies for the OIJ (left), impact of tuples broadcasting (right).

2.4 WindFlow Library

Different parallelization strategies for the OIJ operator will be discussed in this paper, specifically in the context of the C++17 WindFlow header-only library[1] for DSP [10]. In this section, we will introduce the basic features of WindFlow and its runtime system.

2.4.1 Graphs of MultiPipes. WindFlow supports the development of data-flow graphs like in other SPEs like Flink and Storm. The distinguishable feature is that this happens using a programming abstraction called `MultiPipe`. This abstraction implements a logical pipeline of consecutive operators. From a physical perspective, each operator is replicated according to its *parallelism level* (i.e., each replica processes different input tuples arriving at the operator in parallel). Depending on the operator characteristics, the communications between replicas of consecutive operators in the `MultiPipe` can be either *direct* (one-to-one communications only) or *shuffle* (each replica of the left-hand side operator connects to all replicas of the right-hand side operator). Fig. 3 shows an example of an application composed of three `MultiPipes`, where the third one starts with an OIJ operator having two replicas. Outputs of the first `MultiPipe` are tuples of the base

[1]WindFlow is publicly available in GitHub: https://github.com/ParaGroup/WindFlow

stream A, while those of the second `MultiPipe` are tuples of the probe stream B.

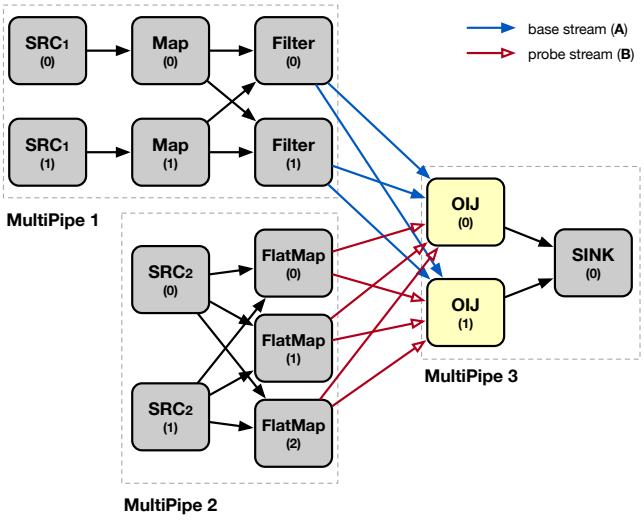

Figure 3. Example of a WindFlow application with the OIJ operator.

It is worth noting that both the base and the probe streams can be concretely implemented by different physical streams. In the figure, the base stream is generated by all the replicas of the Filter operator ending the first `MultiPipe`, so it is physically implemented by two streams per replica of the OIJ. Analogously, the probe stream is implemented by three streams from each FlatMap replica in the second `MultiPipe` (still per OIJ replica).

2.4.2 Runtime system. WindFlow is built on FastFlow[2], a parallel library for efficient pipelining and data shuffling. Each operator's replica runs on a dedicated thread, while replicas of different operators can be executed by a single thread using *chaining*. Data exchange occurs through lock-free Single-Producer Single-Consumer queues, with each position holding a memory pointer to the data (usually heap-allocated). This data can be a single tuple or a batch if batching is used to reduce communication costs. To ease the load on the system memory allocator, WindFlow uses a recycling technique with Multi-Producer Single-Consumer lock-free queues for feedback channels. The FastFlow layer also supports distributed channels for streams connecting threads in different processes via the MCTL message-passing library[4], though such channels are not yet supported by WindFlow.

The general approach of FastFlow is based on the shared-nothing paradigm (i.e., message-passing). While this is obvious for distributed channels, where messages are transmitted by value, the same conceptual paradigm is also adopted for communication between threads of the same process through lock-free queues. In this case, pointers are passed from a producer to a consumer by conceptually releasing the ownership of the data. Therefore, apart from lock-free queues, no data sharing is natively provided by FastFlow.

2.4.3 Out-of-order data streams. Tuples might arrive at operators unordered by timestamp, necessitating stream progress mechanisms for stateful operators like window-based aggregates and joins. WindFlow uses watermarking [1] for this purpose. Receiving a watermark *wm* means no future tuple with a timestamp lower than *wm* can be accepted (otherwise, the tuple is dropped). Unlike SPEs like Storm and Flink, which use user-generated *punctuations* to convey watermarks, WindFlow embeds watermarks within tuples (*pointstamp*). Punctuation messages are automatically generated by the runtime system only when no data is transferred through a stream for a monitored period, and can be manually generated by users at sources during delays. This technique allows watermarks to propagate without additional messages if tuples flow continuously.

3 OIJ Parallel Patterns

The parallel patterns in PPOIJ are based on several choices regarding: *i)* how tuples are distributed from the preceding operator to the replicas of the OIJ; *ii)* the number of OIJ replicas involved in computing the joining pairs of tuples relative to the same key attribute. In all cases, the patterns do not require data sharing and can be easily implemented on single machines as well as in distributed architectures.

3.1 Key-based Parallelism

The **KP-OIJ** pattern, used by many SPEs like Apache Flink, assigns each key to one OIJ replica. Fig. 4 illustrates this concept. Tuples with the same key are routed to the same replica, with point-to-point distribution. The replica maintains state buffers for the base and probe streams of its assigned key, processing tuples in sequence (not necessarily ordered by timestamps). Watermarks purge tuples from state buffers when they are no longer useful.

3.2 Data Parallelism

The second pattern is based on data parallelism. The idea is to partition the state buffers of the probe stream and the base stream for each key among *all* the OIJ replicas. Consequently, each replica owns one partition of the state buffer for both the base and probe streams and for each key. The conceptual steps are as follows: *i)* each tuple is broadcast to all the OIJ replicas; *ii)* each replica, in parallel, evaluates the join condition with all tuples in its partition of the state buffer of the opposite stream; *iii)* only one of the replicas is responsible for storing the received tuple in its partition of the state buffer of the probe (base) stream. Fig. 5 illustrates the core concept of the **DP-OIJ** pattern. The key features of this solution are: *i)* the broadcast distribution of tuples to the OIJ replicas; and *ii)* the partitioning of the state buffers for tuples from both the base and probe streams of each key. Watermarks are

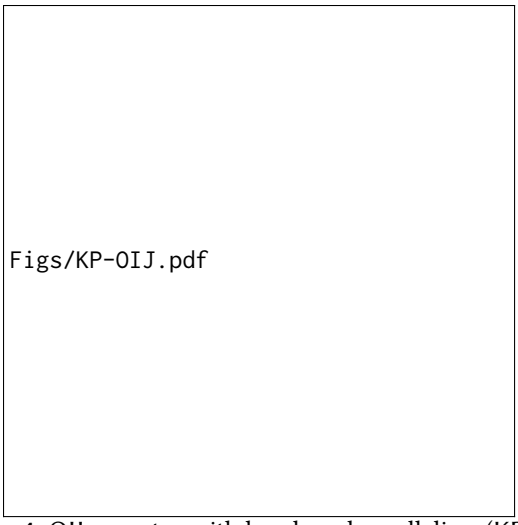

Figure 4. OIJ operator with key-based parallelism (KP-OIJ). Example with two OIJ replicas.

used to purge from the state buffers those tuples that are no longer useful.

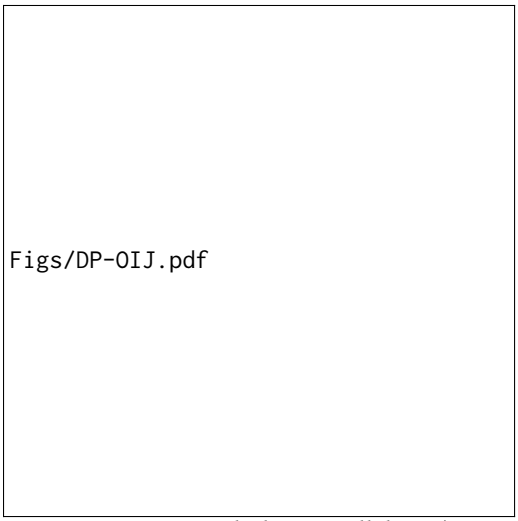

Figure 5. OIJ operator with data parallelism (DP-OIJ). Example with two OIJ replicas.

3.3 Hybrid Parallelism

The hybrid solution combines the low overhead of KP-OIJ (point-to-point distribution) with the improved load balancing of DP-OIJ (equal tuple processing by replicas). The fundamental concept of the hybrid solution (**HP-OIJ**) is to assign each key to a subset of the OIJ replicas, rather than just one (as in KP-OIJ) or all (as in DP-OIJ).

We introduce the *splitting degree* of the key $k \in \mathcal{K}$, denoted by $\mathcal{S}(k) \in [1..N]$, where $N > 0$ is the number of OIJ replicas. This value identifies the number of replicas responsible for computing join pairs for key k. Each key is assigned to a subset of the OIJ replicas. Fig. 6 illustrates this with three replicas handling two keys: the first key is assigned to the first two replicas, and the second key to the last two. State buffers for the same key are partitioned among the assigned replicas and remain private. Watermarks purge tuples from state buffers when they are no longer useful.

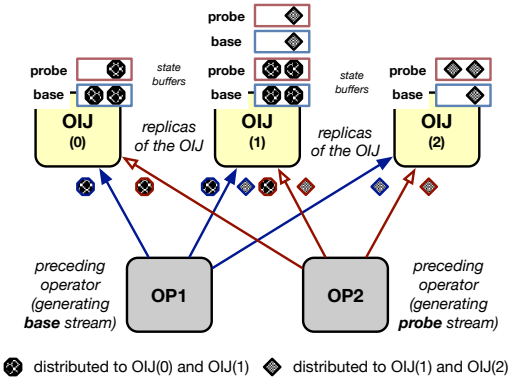

Figure 6. OIJ operator with hybrid parallelism (HP-OIJ). Example with three OIJ replicas.

The assignment of keys is pivotal in obtaining good load balancing with moderate key splitting, if possible. The general idea is to monitor the frequency of the keys at runtime (see § 4) and to assign OIJ replicas to keys. Let $\mathcal{K} = 0, 1, \ldots, K-1$ be the distinct keys, and $\mathcal{N} = 0, 1, \ldots, N-1$ the set of OIJ replicas. Let $f_i \in [0, 1]$ be the relative frequency of the i-th key such that $\sum_{i=0}^{K-1} f_i = 1$. We denote by x_{ij} a binary variable equal to 1 if and only if the i-th key is assigned to the j-th replica, and 0 otherwise. Let y_{ij} be the fraction of the frequency of the i-th key that is assigned to replica j. The problem can be formally stated as follows:

$$\min \max_{j \in N} \left| \sum_{i=0}^{K-1} y_{ij} - \frac{1}{N} \right| \tag{1}$$

subject to

$$\sum_{j=0}^{N-1} x_{ij} \geq 1 \quad \forall i \in [0..K-1] \tag{2}$$

$$\sum_{j=0}^{N-1} y_{ij} = f_i \quad \forall i \in [0..K-1] \tag{3}$$

$$y_{ij} \leq x_{ij} \cdot f_i \quad \forall i \in [0..K-1], \forall j \in [0..N-1] \tag{4}$$

$$y_{ij} = \frac{f_i}{\sum_{j'=0}^{N-1} x_{ij'}} \quad \forall i \in [0..K-1], \forall j \in [0..N-1] \text{ if } x_{ij} = 1 \tag{5}$$

The goal (1) is to minimize the maximum load difference where: (2) each key must be assigned to at least one replica; (3) the entire frequency of each key must be assigned to replicas; (4) each replica can be assigned to a fraction of

the frequency of each key; (5) if a key is assigned to more replicas, each will receive an even fraction of its frequency.

The combinatorial optimization problem described above is NP-hard, since it can be considered a variation of the multi-way partition problem, which aims to split a multi-set of numbers so each partition's sum is approximately equal. Our problem adds complexity by allowing the same key to be assigned to multiple OIJ replicas, evenly splitting the volume of tuples of that key. Balancing the load while minimizing splits (to reduce communication overheads) is complex, as it requires minimizing the number of replicas per key while ensuring no replica has an excessive load. We propose a heuristic solution in Alg. 1. The algorithm takes as input a threshold $\theta \geq 1$, an array f of frequencies (one per key), a key-to-replicas mapping *key2R*, and a maximum splitting degree *max_splitting* that can be utilized by the heuristics.

Algorithm 1: Assign keys to OIJ replicas

Input: θ, f, *key2R*, *replicas*, N
Output: Balanced assignment of keys to OIJ replicas

```
1  Initialize split with zeros of size K;
2  Initialize keys with (i, f[i]) for i ∈ [0..K − 1];
3  Sort keys in descending order of frequencies;
4  for each (i, f[i]) in keys do
5      Find minR with minimum load in replicas;
6      minR.load ← minR.load + f[i];
7      split[i] ← 1;
8      key2R[i].push_back(minR);
9  if isBalanced(replicas, θ) then
10     return;
11 balanced ← false;
12 while not balanced do
13     for each (i, f[i]) in keys do
14         if split[i] < N then
15             Find minR in replicas with minimum load that
                 does not have key i;
16             for each replica j in replicas already assigned
                 to key i do
17                 j.load ← j.load − (f[i]/split[i]) +
                     (f[i]/(split[i] + 1));
18             minR.load ← minR.load + f[i]/(split[i] + 1);
19             split[i] ← split[i] + 1;
20             key2R[i].push_back(minR);
21             if isBalanced(replicas, θ) then
22                 balanced ← true;
23                 break;
24     if isBalanced(replicas, θ) || all_max_split_reached then
25         balanced ← true;
26 return;
```

Each key is initially assigned to the least loaded replica. If the load exceeds a threshold, the algorithm redistributes keys among replicas until balance is achieved or the maximum splitting degree is reached. During this phase, the same key might be assigned to multiple replicas, evenly splitting the load. The `isBalanced` function checks if the ratio of the maximum load to the average load exceeds a threshold θ, returning *false* if it does, and *true* otherwise. It also ensures all replicas receive a non-zero load.

4 Implementation

The WindFlow implementation[2] of the three parallel patterns uses the shared-nothing paradigm. Each OIJ replica has two private state buffers per key: one for the probe stream and one for the base stream. It also maintains a private hash table (an `std::unordered_map` from the C++ STL) that maps keys to a descriptor structure with pointers to the state buffers, providing average constant-time key searches. Buffers, ordered by timestamp, facilitate logarithmic complexity look-ups via binary search. They are periodically purged of old elements based on received watermarks. Since old tuples are removed from the beginning of the buffer, it is implemented as a double-ended queue (`std::deque`).

Although no race conditions and synchronization are needed to manipulate state buffers, the complexity of the shared-nothing implementation is concentrated in the distribution and collection of tuples. In WindFlow such activities are performed by *Emitters* (`E`) and *Collectors* (`C`) functionalities as depicted in Fig. 7.

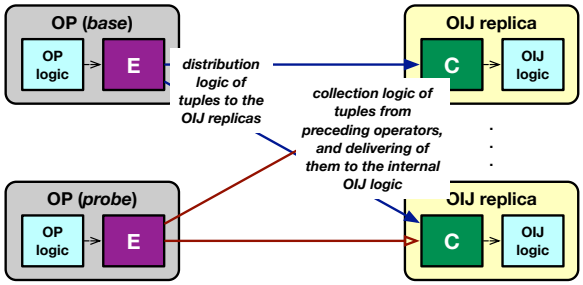

Figure 7. Shared-nothing implementation of the OIJ.

4.1 KP-OIJ Implementation

Each emitter distributes tuples to the OIJ replicas by delivering all tuples with the same key to the same replica. This is done by computing the key's hash value and mapping it to the destination replicas' index range. In batching mode, the emitter maintains a partial batch per destination and fills it based on the tuple's key. Once filled, the batch is transmitted. Before allocating a new batch, the emitter tries to recycle an existing one from a feedback lock-free queue where processed batches can be reused without re-allocating memory

[2]The source code of the patterns has been partially integrated with WindFlow, and it is available in GitHub: https://github.com/DropB1t/WindFlow.

from scratch (see § 2.4). This recycling approach is adopted by all patterns of this section.

The collector receives tuples non-deterministically from any input stream and delivers them in any order to the component processing the OIJ logic. While any ordering is tolerated, watermarks must be properly propagated downstream. This is achieved by maintaining the maximum watermark received from each input stream and calculating the minimum among them.

4.2 DP-OIJ Implementation

The DP-OIJ implementation requires specific emitter and collector functionalities. Each tuple must be delivered to all OIJ replicas, necessitating *broadcast* communication, which is performed by value on distributed channels. For channels between threads of the same process, the tuple or batch to be delivered to multiple destinations is passed using a shared pointer. This approach grants read-only access to the destinations and uses an atomic counter to ensure the tuple/batch is correctly destroyed (actually recycled) after all destinations have read it.

The collection phase requires special care. The nondeterministic arrival order from multiple input streams might result in incorrect behavior that violates OIJ computational semantics. Consider an OIJ operator with two replicas (R_0 and R_1). Let a_1 and b_1 be two input tuples from the base and probe streams, respectively. Assuming the pair (a_1, b_1) needs to be generated, two incorrect behaviors might occur if the collection phase is handled as in KP-OIJ.

Example 4.1 (Duplicate join pairs)**.** This scenario is illustrated in Fig. 8 as a sequence diagram. Tuples a_1 and b_1 are sent to both replicas but received in different orders. R_0 receives a_1 first, suppose it is the designed replica to save a_1 based on its hash, then receives b_1 and emits (a_1, b_1). Meanwhile, R_1 receives b_1 first, suppose it is designated to saves it, then receives a_1 and also emits (a_1, b_1).

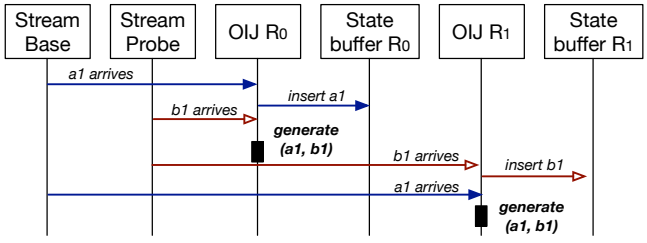

Figure 8. Duplicate output results.

Example 4.2 (Missing join pairs)**.** This scenario is shown in Fig. 9. R_0 receives b_1 first but does not save it, then receives a_1 and saves it. No pair (a_1, b_1) is emitted. Similarly, R_1 receives a_1 first but does not save it, then receives b_1 and saves it. No pair (a_1, b_1) is emitted.

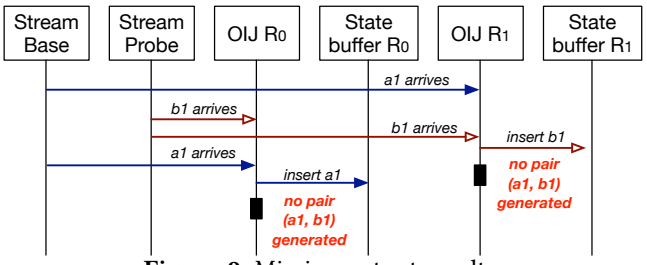

Figure 9. Missing output results.

Both scenarios share the same issue: tuples are received by the OIJ replicas in a different order. To address this, we implemented a specific collector for DP-OIJ. This collector uses a static round-robin ordering of input streams, alternating between a probe stream and a base stream. It buffers tuples received from a different stream relative to the expected one according to the circular ordering. The same behavior is adopted if the transmission occurs in batches. Watermarks are correctly propagated by considering the minimum watermark among the internal queues (one per stream).

4.3 HP-OIJ Implementation

HP-OIJ associates a different splitting degree per key based on the estimated frequencies. This solution shares some challenges about tuples' ordering with the DP-OIJ solution. However, a fixed round-robin ordering of input streams in the collector is no longer sufficient to guarantee correctness.

Example 4.3 (Wrong ordering of tuples)**.** Suppose two OIJ replicas, R_0 and R_1, receive tuples from emitters E_0 and E_1 (base and probe streams). Keys k_0 and k_1 are assigned to R_0, with k_1 also assigned to R_1. The replicas alternate reading from E_0 and E_1. E_0 delivers t_1 of key k_0 to R_0, then t_3 of key k_1 to both replicas. E_1 delivers t_2 of key k_1 to both replicas. Thus, the replicas read key k_1 tuples in different orders.

To avoid the aforementioned problem, HP-OIJ needs a careful implementation:

- emitters must distribute tuples to the destination replicas based on the current assignment obtained with Alg. 1. If batching is enabled, the emitter must prepare batches of tuples all having the same key (while in KP-OIJ and DP-OIJ batches may contain tuples of different keys);
- collectors must guarantee that the replicas associated with a given key will receive tuples of that key in the same order. Therefore, the condition already applied to collectors in DP-OIJ is now on a *key-basis*, needing a buffer of tuples per input stream an for each distinct key, i.e., $O(K \cdot M)$ buffers with K the number of keys and M the number of emitters sending tuples to each OIJ replica (this matches the number of replicas of the two preceding operators in the data-flow graph).

In HP-OIJ, watermarks are also considered on a key-basis. The watermark conveyed by a tuple is used by the OIJ replica to purge old tuples from the state buffers of that key only, as watermarks of different tuples are no longer monotonically increasing in this parallel pattern. To emit watermarks correctly, each join pair emitted by an OIJ replica will incorporate a watermark equal to the minimum of the last watermarks received by the same replica across all keys.

A final, fundamental concern of HP-OIJ is that Alg. 1 requires prior knowledge of the key frequencies. To address this issue, HP-OIJ behaves as KP-OIJ during and initial period of the execution that we call *calibration phase*. Each emitter monitors its transmitted tuples' distribution. After calibration, emitters send their local distributions to the first OIJ replica (master), which computes the global distribution and applies Alg. 1. The resulting key-replica mapping is transmitted through feedback channels (see § 2.4) to the emitters, allowing on-the-fly usage without data stream interruption. For correctness, a replica assigned to a key during calibration must remain assigned (possibly with others) after Alg. 1. Thus, lines 4-10 are skipped when HP-OIJ is configured to estimate key frequencies at runtime.

4.4 Final Comparison

To elucidate the implementation differences among the three presented patterns, we present in Tab. 1 the implementation strategies for the emitters and collectors, as well as the watermarking strategy adopted by each pattern.

Pattern	Emitters	Collectors	Watermarks
KP-OIJ	Point-to-point distribution	Standard collector multiplexing input streams	Global low-watermarks
DP-OIJ	Full-broadcast distribution	Round-robin gathering policy from input streams	Global low-watermarks
HP-OIJ	Partial-multicast distribution	Round-robin gathering policy per distinct key	Key-based low-watermarks

Table 1. Summary of the implementation characteristics of the OIJ patterns.

5 Evaluation

This section is devoted to presenting the performance assessment of the parallel patterns for OIJ introduced before. Our goal is to highlight their *pros* and *cons*, and the effectiveness of the hybrid solution in several practical scenarios.

Hardware. Our experiments were conducted on a server equipped with an AMD EPYC 9534 CPU, featuring 64 cores operating at a base frequency of 2.45 GHz and boosting up to 3.7 GHz with turbo boost. The CPU includes an L3 cache of 256 MiB, while each core has an L1 cache of 32 KiB for instructions and 32 KiB for data, and an L2 cache of 1 MiB. Simultaneous Multithreading (SMT) was disabled during the experiments. The machine runs Ubuntu 22.04.5 LTS.

Tools and compilers. This work utilizes WindFlow version `4.2.0`, compiled with `gcc` version `12.3.0` at the highest optimization level (flag `-O3`). Additionally, for comparison we employ Flink version `1.16.3` with Java `11.0.25`. Each test is conducted five times, and whenever possible, the 95^{th} confidence interval is displayed as error bars in the plots.

Workloads. The query consists of two source operators producing the base and probe streams, respectively, which feed into the OIJ operator. Results are transmitted to the final Sink operator. The sources are designed to generate a parameterized workload. The input tuples are generated based on the Stock and Rovio datasets from the AllianceDB benchmark suite [19], which provides real-world traces related to financial markets and advertisement campaigns. The sources can generate streams with a configured number of distinct keys and a parameterized distribution. We consider two scenarios: *uniform distribution* and *skewed distribution*. The latter is obtained using a self-similar distribution with a given skew factor s in the range $(0, 1)$ (the smaller the value, the more skewed the distribution). Such a distribution has been considered realistic of the skewness of real-world streaming workloads [8]. The input stream is generated at the fastest sustainable speed by the sources. The OIJ uses a controllable interval I expressed in seconds (converted into two time boundaries `lwr` and `upr`, see § 2).

Baselines. We compare the three patterns (KP-OIJ, DP-OIJ, and HP-OIJ) in WindFlow among themselves. Additionally, we provide a comparison against the OIJ operator implemented by Flink. We also conduct an experimental comparison against the approach named Scale-OIJ [18], which shares some similarities with our research, although it is based on a different design and implementation (i.e., shared memory and lock-free concurrent data structures shared by replicas).

Evaluation metrics. Our evaluation focuses on two performance metrics: *throughput* and *latency*. Throughput is measured as the average number of input tuples the query can process per second. Latency is the end-to-end elapsed time from when a tuple is materialized in the source to when the corresponding output result is received by the sink. To achieve this, each join pair incorporates a timestamp that is the maximum of the timestamps of the two tuples (from the base and probe streams, respectively) that compose the pair.

5.1 Performance Comparison

In this section, we assess the impact of various parameters on the performance of the three patterns to understand the achievable trade-offs.

Impact of the number of keys and distribution. Fig. 10 shows the throughput of the three patterns with varying

key numbers K under a uniform distribution. Results are reported for intervals $I = 10$ seconds (left) and $I = 30$ seconds (right). Experiments used eight OIJ replicas and batches of 32 tuples. With a uniform distribution, load balancing is straightforward if enough keys exist. For $K \geq 25$, KP-OIJ provides higher throughput due to its simple runtime implementation. With very few keys, KP-OIJ fails to balance the load, leaving some replicas idle. In such cases, HP-OIJ achieves higher throughput. Although DP-OIJ achieves perfect load balancing, its runtime overheads with full broadcast make it useful only in the extreme case of 1 key.

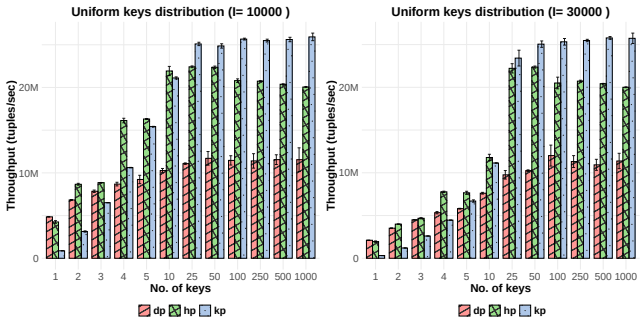

Figure 10. Throughput with different number of keys (uniform distribution) and OIJ intervals I.

Fig.11 shows results with a skewed key distribution generated with a moderate skew factor of 0.35. The experiment indicates that DP-OIJ is the best solution with a small number of keys, as high key splitting is necessary to balance the load and achieve high throughput. With more keys, HP-OIJ is optimal, balancing the load with moderate splitting and lower runtime overheads for distributing tuples. Except in a few cases with many keys, KP-OIJ is generally ineffective with skewed distributions.

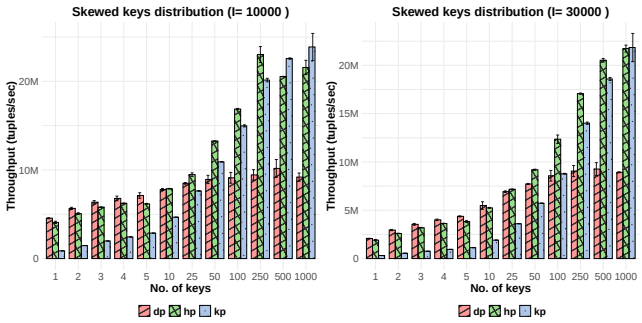

Figure 11. Throughput with different number of keys (skewed distribution, $s = 0.35$) and OIJ intervals I.

To better highlight HP-OIJ's behavior in a skewed scenario, Fig.12(left) reports the average splitting degree adopted with hybrid parallelism for different numbers of keys. As expected, with few keys available, the splitting degree is almost at its maximum, similar to DP-OIJ. However, it is greatly reduced when more keys are available. This showcases HP-OIJ's ability to adapt, mimicking DP-OIJ when few keys are present and reducing splitting as more keys become available, based on their measured frequency distribution. Fig.12(right) illustrates the load balancing factor (where 1 indicates perfect balancing). While DP-OIJ consistently achieves perfect balancing by definition, KP-OIJ approximates the best scenario when many different keys exist in the input stream. Meanwhile, HP-OIJ achieves load balancing nearly identical to DP-OIJ, confirming the effectiveness of the heuristics described in Alg. 1.

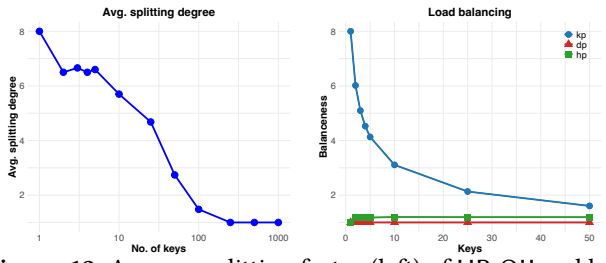

Figure 12. Average splitting factor (left) of HP-OIJ and balanceness of the different patterns (right).

How the number of OIJ replicas impacts performance. Changing the number of replicas impacts the throughput of OIJ. Fig. 13 shows results with a skewed distribution, 50 distinct keys, and two intervals, varying the number of replicas for the three patterns. HP-OIJ achieves better scalability in both cases, while KP-OIJ is limited by the skewed distribution and DP-OIJ by its runtime overheads. Increasing parallelism beyond the maximum shown yields only small performance improvements.

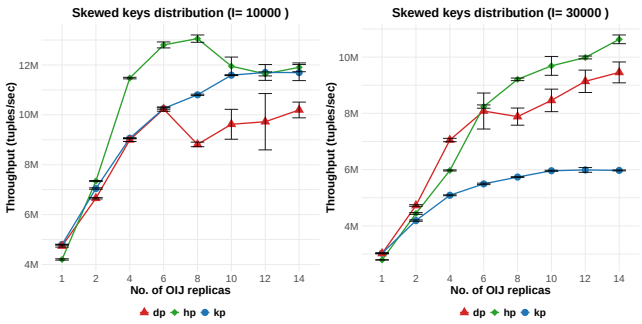

Figure 13. Scalability (skewed distribution, $s = 0.35$) and different OIJ intervals I.

How the skewness of the input stream impacts performance. We previously adopted a skewed distribution with a moderate factor of 0.35. To assess performance under varying skewness, we analyzed different skewness factors (lower values indicate more skewness, with 0.5 representing a uniform distribution). Fig. 14 shows results with two intervals

and parallelism fixed at eight for all patterns. As observed, DP-OIJ's performance is almost independent of the skew factor, as it uses the all-tuple-to-all-replicas paradigm regardless of frequency distribution, resulting in evenly sized state buffers among OIJ replicas. KP-OIJ performs better with less skewness, improving load balancing by assigning each key to one replica. HP-OIJ is slower than DP-OIJ with extreme skewness, as its splitting degree is nearly maximum and its more complex implementation (§ 4.3) results in higher overheads when mimicking DP-OIJ. Conversely, with low skewness, HP-OIJ is slightly worse than KP-OIJ but clearly superior in intermediate skewness scenarios.

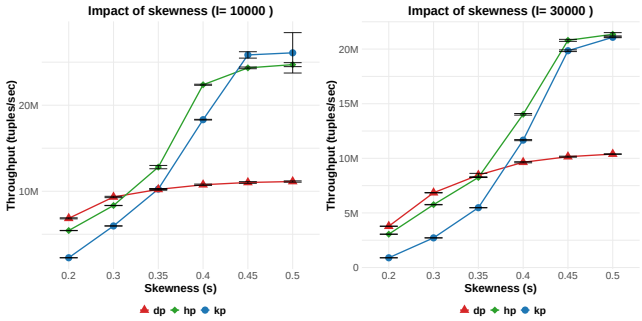

Figure 14. Impact of skewness in the keys distribution.

How patterns benefit from batching. Fig.15(left) and Fig.15(right) show the impact of batching with $I = 30$ seconds, eight replicas, and 50 keys. Without batching (i.e., $b = 1$), KP-OIJ performs best in both scenarios. This is expected with a uniform distribution, as KP-OIJ achieves good balancing and high throughput. However, with a skewed distribution ($s = 0.35$), small batching is crucial to amortize communication overheads for both DP-OIJ and, to a lesser extent, HP-OIJ, where the same tuple might be sent to multiple replicas. In the skewed case, a very small batch of 2 tuples allows HP-OIJ to outperform KP-OIJ due to better load balancing.

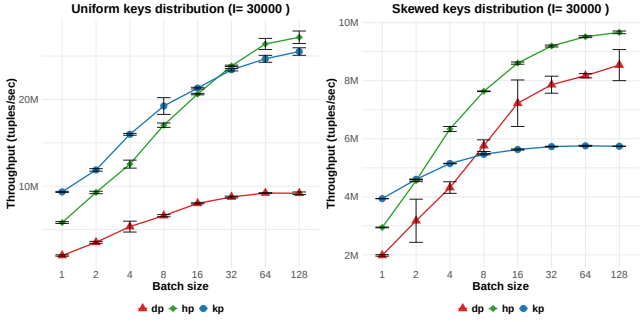

Figure 15. Impact of batching.

5.2 Latency Analysis

We evaluate latencies under two distribution scenarios: uniform and skewed ($s = 0.35$). Using an interval of $I = 10$ seconds and eight replicas for all patterns, Fig. 16 shows the CDF of latencies collected during the tests. In the uniform distribution scenario (left), HP-OIJ uses minimal splitting to balance the load, resulting in latencies similar to KP-OIJ. Here, DP-OIJ has a higher average latency of 27.9 ms compared to HP-OIJ's 2.05 ms. With moderate/low splitting, HP-OIJ maintains low latencies as the buffering delay in the collectors (with separate queues per key and channel) is almost zero.

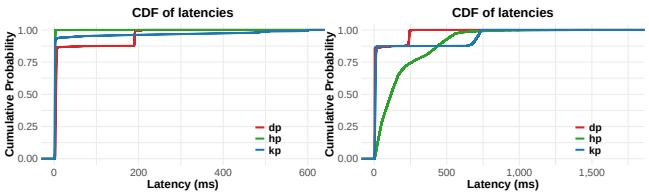

Figure 16. Latency with uniform distribution (left) and skewed distribution (right).

Figure 16(right) presents the results of the same experiment with a skewed distribution. Interestingly, DP-OIJ achieves low latency values, with a mean of 33.2 ms and a p95 latency of 241.1 ms. This is due to DP-OIJ's ability to perfectly balance the size of the state buffers, which are equally partitioned among the replicas. Consequently, the latency to produce a single join pair, computed by one replica, is proportional to the size of the state buffers maintained by that specific replica. In contrast, KP-OIJ exhibits high latency due to significant load imbalances, with a mean latency of 94.8 ms and a p95 latency of 714.7 ms. HP-OIJ shows the worst latency results in this scenario, with a mean latency of 111.2 ms and a p95 latency of 828.6 ms, which is significantly worse than DP-OIJ and slightly worse than KP-OIJ. Upon further investigation and profiling, we discovered that the higher latencies are primarily caused by the time tuples spend in the collector queues. The size of these queues grows with the skewed distribution, as the collectors must ensure the same ordering of tuples with the same key across different replicas. This is the latency cost associated with hybrid parallelism under the shared-nothing paradigm.

5.3 Comparison with State-of-the-Art

In this final section, we present the results of our comparison with Flink, an open-source SPE, and the research prototype Scale-OIJ [18].

Flink is an open-source JVM-based SPE with a large user community. It offers features like transparent fault-tolerance, scaling out, and integration with various external messaging,

log, and database systems, though these can impact performance. The OIJ operator in Flink uses internal parallelization similar to the KP-OIJ pattern described in this paper.

Scale-OIJ is a prototype introduced in [18] and is part of OpenMLDB, a machine learning database written in C++. It shared some ideas used in our hybrid parallelism approach. However, Scale-OIJ is based on a shared-memory implementation, utilizing concurrent data structures to manage state buffers shared by the threads running the OIJ replicas. Further details on its design and the differences from our work are discussed in § 6. Our paper does not include a direct comparison with SplitJoin [11], as Scale-OIJ has already demonstrated superior performance, as detailed in [18].

For the comparison, we adapt the benchmark from the Scale-OIJ repository. This benchmark generates tuples with a key of type `std::string` synthetically, and other fields of types `int` and `std::string`. The number of keys and the parallelism degree of the OIJ implementation are configurable command-line parameters. We use the authors' *dynamic* scheduling to balance the workload by allowing more replicas to access the state buffers of the same key concurrently. Due to the more complex tuple generation and joining conditions, the overall throughput is lower than in the previous experiments. We adapt the same benchmark to run with WindFlow using the HP-OIJ pattern and develop an equivalent implementation for Flink based on a Java translation of the code.

Throughput. Fig. 17 presents the experimental results for 100 keys (left) and 1000 keys (right). Since both Scale-OIJ and Flink process data one tuple at a time, we compare two versions of WindFlow: one with batching disabled (`windflow_b1`) and the other with the default batching value used throughout the paper (`windflow_b32`).

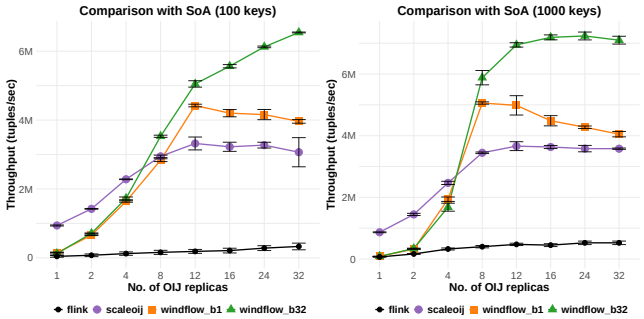

Figure 17. Throughput comparison between WindFlow, Flink and Scale-OIJ.

Scale-OIJ, due to its shared-memory implementation using concurrent data structures, demonstrates better throughput in both scenarios with a low number of OIJ replicas (fewer than 4). However, with increased parallelism, WindFlow utilizing the HP-OIJ pattern outperforms Scale-OIJ both without and with batching enabled. The latter achieves higher peak throughput, allowing for better scalability as communication costs are more effectively amortized. While Scale-OIJ avoids explicit communication overheads through shared-memory parallelism, the concurrent access to shared data structures by replicas (despite being lock-free) does not yield benefits in the considered scenario. Flink exhibits a much lower throughput, scaling better with more keys due to a better assignment of keys to replicas.

Latency. Fig. 18 illustrates the latency results from the previous experiment, which involved 100 keys and a parallelism level of 8. In this analysis, we exclude the latencies recorded by Flink due to their higher values, which would make the plot less comprehensible. Both Scale-OIJ and WindFlow (without batching) achieved similar throughput, while WindFlow (with batching) outperformed the other two. As observed, the latency distributions for Scale-OIJ and WindFlow (without batching) are quite similar. However, enabling batching in WindFlow results in slightly higher latency values, with a maximum bounded at less than 27 ms.

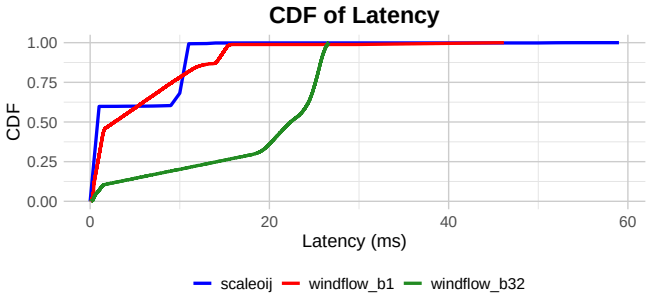

Figure 18. Latency comparison between WindFlow and Scale-OIJ.

6 Related Works

Stream joins are challenging operators that can bottleneck continuous queries. Their stateful nature introduces semantic issues, requiring restrictions on stream portions to identify valid join pairs. One solution is using sliding windows [15], which consider the most recent tuples. Variations like interval joins are also studied. Kang [7] proposed a general algorithm for stream joins, involving processing a tuple by scanning the opposite stream's window, inserting the new tuple, and removing expired tuples.

Parallelization techniques for stream joins have been extensively studied. Handshake join [14] uses a pipeline of workers exchanging tuples in opposite directions, evaluating the join condition when a pair arrives. However, its high latency depends on the streams' arrival rates. A solution [12] proposed a forwarding approach for streams with low or intermittent speeds. CellJoin [6], designed for the IBM Cell multiprocessor, uses a shared-memory implementation with dynamically partitioned stream windows among workers. Research has also focused on novel algorithmic solutions,

such as chain index [9] and PIM tree [13], rather than parallel solutions. The integration of efficient data structures is orthogonal to our work, which focuses on parallelization patterns. The shared-nothing paradigm adopted in our work facilitates the use of different data structures for state buffers, as they do not need to be concurrent and thread-safe.

SplitJoin [11] is closely related to our DP-OIJ pattern. In SplitJoin, each tuple is broadcast to all replicas of the join operator, which compute the join pairs with that tuple in parallel by accessing their partition of the state buffer. Despite its similarities to our DP-OIJ pattern, SplitJoin has several limitations that our work addresses. Firstly, to ensure correctness, as discussed in § 4.2, SplitJoin requires a strict ordering of input tuples. This is achieved using a distribution tree for delivering tuples to replicas and another tree for collecting results, executed by dedicated distributor entities running the internal tree nodes. This approach increases thread oversubscription and core consumption. Secondly, SplitJoin assumes that tuples from both streams are provided to the OIJ operator through a single physical stream. However, in recent SPEs, multiple replicas of preceding operators can produce tuples at high speed, as illustrated in Fig. 3.

Particularly important in our analysis was Scale-OIJ [18], which focused on efficient OIJ implementations. The idea underpinning their work has some analogies with our HP-OIJ pattern: assigning one key to only an OIJ replica might result in load imbalance, so more replicas can contribute to processing tuples of the same key. However, the Scale-OIJ approach adopts shared-memory parallelization, which is feasible only within a multicore. State buffers are implemented as concurrent data structures, while tuples are distributed to one replica only. A replica receiving a tuple must access the partition of the state buffers of the same key owned by other replicas. Such concurrent accesses, although lock-free, limit the performance of this solution as described in § 5.3. Furthermore, this solution is not applicable in distributed SPEs without involving external data storage, which is detrimental to performance. Additionally, the ability to process out-of-order data streams in Scale-OIJ is achieved by expressing a user-defined maximum lateness, without adopting streams with watermarks, which represent the de-facto solution for out-of-order data streams in modern SPEs.

7 Conclusions

This paper explores parallel patterns for online interval joins on data streams, focusing on shared-nothing parallelizations. We propose a hybrid parallelism approach with the HP-OIJ pattern, where key assignment to OIJ replicas is determined after an initial calibration phase. Keys can be assigned to multiple replicas to balance the computational load. We examine the implementation and correctness requirements. Experimental analysis shows our solution's effectiveness compared to state-of-the-art methods. Future work includes implementing our patterns in other SPEs like Flink and studying performance in distributed architectures.

Acknowledgments

This work was partially supported by the Italian PRIN project OUTFIT (2022BAL2F3), and by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP C59J24000110004, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

References

[1] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow. *Proc. VLDB Endow.* 14, 12 (July 2021), 3135–3147. https://doi.org/10.14778/3476311.3476389

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multicore.* John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13

[3] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics* (1st ed.). Cambridge University Press, USA.

[4] Federico Finocchio, Nicoló Tonci, and Massimo Torquati. 2024. MTCL: A Multi-transport Communication Library. In *Euro-Par 2023: Parallel Processing Workshops: Euro-Par 2023 International Workshops, Limassol, Cyprus, August 28 – September 1, 2023, Revised Selected Papers, Part I* (Limassol, Cyprus). Springer-Verlag, Berlin, Heidelberg, 55–67. https://doi.org/10.1007/978-3-031-50684-0_5

[5] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (Nov. 2023), 507–541. https://doi.org/10.1007/s00778-023-00819-8

[6] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB Journal* 18, 2 (April 2009), 501–519. https://doi.org/10.1007/s00778-008-0116-z

[7] J. Kang, J.F. Naughton, and S.D. Viglas. 2003. Evaluating window joins over unbounded streams. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405).* 341–352. https://doi.org/10.1109/ICDE.2003.1260804

[8] Flip Korn, S. Muthukrishnan, and Yihua Wu. 2006. Modeling skew in data streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06).* Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/1142473.1142495

[9] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15).* Association for Computing Machinery, New York, NY, USA, 811–825. https://doi.org/10.1145/2723372.2746485

[10] Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, and Marco Danelutto. 2021. WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2748–2763. https://doi.org/10.1109/TPDS.2021.3073970

[11] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2016. SplitJoin: a scalable, low-latency stream join architecture with adjustable ordering precision. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16).* USENIX Association, USA, 493–505.

[12] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proc. VLDB Endow.* 7, 9 (May 2014), 709–720. https://doi.org/10.14778/2732939.2732944

[13] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2523–2537. https://doi.org/10.1145/3318464.3380576

[14] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 625–636. https://doi.org/10.1145/1989323.1989389

[15] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2022. Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems. *Datenbank-Spektrum* 22, 2 (2022), 99–107. https://doi.org/10.1007/s13222-022-00417-y

[16] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *The VLDB Journal* 32, 5 (Feb. 2023), 985–1011. https://doi.org/10.1007/s00778-022-00778-6

[17] Junyi Xie and Jun Yang. 2007. *A Survey of Join Processing in Data Streams*. Springer US, Boston, MA, 209–236. https://doi.org/10.1007/978-0-387-47534-9_10

[18] Hao Zhang, Xianzhi Zeng, Shuhao Zhang, Xinyi Liu, Mian Lu, and Zhao Zheng. 2023. Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3031–3042. https://doi.org/10.1109/ICDE55515.2023.00232

[19] Shuhao Zhang. 2024. AllianceDB. https://github.com/intellistream/AllianceDB. [Online; accessed 25. Sep. 2024].