Conflux: Order Preservation in Distributed Stream Processing

Yuechuan Zhang*
*Department of Computer Science
University of Illinois Urbana-Champaign
Email: yz134@illinois.edu

Sarthak Singh†, Feize Shi†
†Department of ECE
University of Illinois Urbana-Champaign
Email: {singh94, feizes2}@illinois.edu

***Abstract*—With the aid of a fast-growing cloud computing infrastructure, distributed and parallel stream processing systems demonstrate their powerful real-time data processing capabilities. While modern stream processing frameworks offer high scalability and throughput through parallel computation, there is a lack of systematic approach for efficient order preservation. This is particularly crucial in contexts where downstream applications, such as fraud detection and trade compliance monitoring, rely on the causal order implied by the source stream. In this paper, we observe the common phenomenon of source stream order disruption intrinsic to parallel stream computation and present Conflux, a novel technique to retain source stream order in system output without restricting parallelism. Our implementation of Conflux over Apache Flink and Apache Kafka demonstrates a 36% reduction in end-to-end latency and a 40% higher throughput compared to a window-based sorting baseline when deployed on a cluster of 20 nodes.**

I. Introduction

A. Motivation

In the realm of stream processing, it is observed that the order of the source stream often carries crucial information about event order and causal relations. This order can be highly beneficial and, in some cases, necessary for downstream applications such as fraud detection services and trading compliance monitors, which aim to identify specific patterns. One motivating real-world example that inspired this paper is a stream processing pipeline that parses network captures of market data emitted from the exchange network into an internally usable format and outputs it to a downstream trading compliance application, which monitors for illegal market manipulations. In this case, the trade monitor assumes event time ordering of the input stream, and a parallel stream processing pipeline given its intrinsic nature of order disruption, easily breaches such SLA. This undesired property can be seen from a simple example. As shown in Fig 1, record X arrives before record Y at a pipeline containing a map transformation with a parallelism of 3. An order inversion happens when record X is processed at a slower rate at map instance 1 and thus emitted to the sink later than record Y.

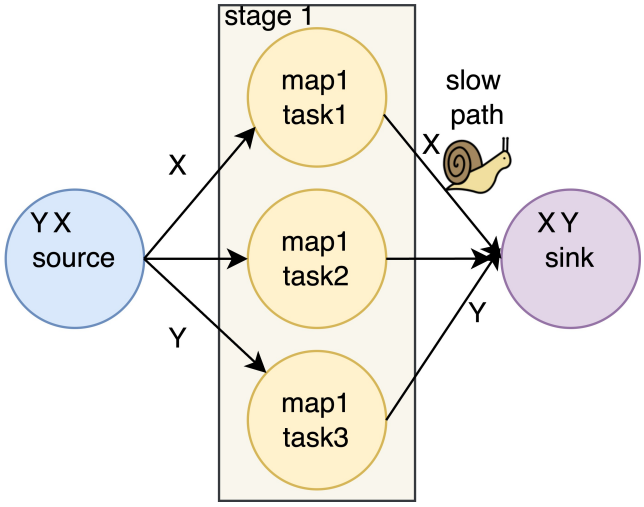

Fig. 1. An example of order disruption in a parallel stream processing pipeline

In the scenario of the trading infrastructure, two possible remedies include re-implementing a monitor that tolerates disordered input streams and performing a real-time global stream sort at the end of the parser pipeline. However, both options are less than ideal: disordered event streams add significant state management overhead and degrade service quality for pattern detection, while a global

sort is extremely expensive, both in terms of the computational complexity and the space overhead for buffering disordered records.

Moreover, a sort buffer local to the sorter inevitably translates to large operator state and system checkpoint burdens when it comes to providing fault tolerance. A third alternative is to restrict the parallelism of the parser pipeline to one. While this trivially preserves the input order, it usually fails to meet system latency requirements. Therefore, creating an efficient source-order-preserving pipeline without harming system parallelism and incurring costly online stream sorting becomes an interesting problem.

B. Methodologies

Our approach is based on the observation that records within a data path could retain their relative ordering under reasonable assumptions. A data path starts from a data source, travels through the job graph, and ends in a data sink. When both the parallel instances of the operators and the communication channels retain FIFO semantics, input records that pass through the same data path preserve their relative order. By exploiting the sorted nature of each data path, we could reduce a global sorting problem into a k-way merge problem and significantly reduce the sorting overhead.

C. Challenges

Reducing a global stream sort to a k-way merge problem poses two significant challenges. Firstly, it necessitates a generic approach to accurately compute the number of data paths for any given pipeline topology. This is particularly challenging due to the diverse collection of stream partitioning semantics available in mainstream stream processing frameworks. Underestimating the number of data paths can lead to a safety violation, compromising the sorted property of each data path by inducing an incorrect partitioning strategy. Secondly, there is a need for a mechanism to prevent temporarily vacant data paths from stalling the k-way merge algorithm, since when dealing with unbounded streams, the algorithm must be able to observe the next record's priority on each data path to determine the global minimum. These two challenges are discussed in detail in Section IV-D and Section IV-E respectively.

II. Related Work

A. Order Preservation with Existing Framework Supports

We dive into the documentation of several widely used open-source stream processing systems, such as Spark Structured Streaming and Flink, in search of solutions that can be developed from existing framework supports. In general, we observe two possible approaches. The first involves sorting based on Window, while the second relies on sorting based on state buffers. As of version 1.17.2, Flink DataStream API supports a tumbling window assigner that breaks a stream into finite chunks based on watermark [1]. Here, a local watermark is derived by taking the minimum of watermarks received on all inbound channels. When a watermark x is received at the window-based sorter, it indicates that all records with a logical timestamp smaller or equal to x have arrived. Similar window semantics are also supported in other open-source stream processing framework APIs, such as Spark Structured Streaming [2]. However, significantly delayed records can prevent the watermark from progressing, creating heavy back pressure [3] that throttles pipeline throughput. In fact, we implement our baseline pipeline based on this solution to showcase its performance against Conflux.

Another approach generally applies to any stream processing framework that supports stateful computations. It involves implementing a stateful sorter that buffers and sorts records on the fly. While the implementation of the state buffer and sorting algorithm may vary, this approach inevitably leads to heavy state management and system snapshot overhead as the local buffer grows under a high-volume stream. In Flink version 1.17.2, one can trade throughput for smaller memory footprint by switching from a memory-based HashMapStateBackend to a file-system-based EmbeddedRocksDBStateBackend [4]. However, as most stream processing frameworks aim to provide robust fault tolerance through global snapshots [5], large operator states can significantly impede snapshot progress and result in long system halts. Therefore, we do not consider existing mainstream framework supports to be effective alternatives to our problem.

B. Other Order Manipulation Techniques

A lot of work also attempts to extend sorting algorithms to distributed scenarios. For example, Jeon et al. consider extending merge sort [6] and bitonic sort [7] to multi-core scenarios. However, similar approaches are more likely to be considered for shared-memory, multi-core scenarios on a single machine. In contrast to merge-based sort, often used in memory or on a single machine, partition-based kind is more suitable for distributed memory environments because it reduces the transfer overhead. Therefore, some work [8]–[10] focused on finding better partitioning methods to accelerate distributed sorting. However, in stream processing scenarios, where the data is unbounded, many sorting algorithms do not apply to our scenario.

III. Problem Model

In this section, we explain the stream processing model that we will work with and describe the preliminaries of our problem setup.

A. Stream Graph

We present a logical representation of a stream processing pipeline as a directed graph consisting of source nodes, transformation nodes, sink nodes, and stream edges. The source nodes are the vertices with zero in-degree as they read from components outside of the system and generate input streams. The transformation nodes read one or more streams (in-bound stream edges) generated from either source nodes or other transformation nodes and produce one or more output streams (outbound stream edges). Finally, the sink nodes accept a stream from a transformation node and output to components outside of the system. Each node in a stream graph carries a parallelism factor that decides how many parallel tasks are required for performing the execution. A stream edge represents a data dispatch pattern from an upstream node to a downstream node. Some common patterns include shuffling, round-robin, and broadcast. Such edges determine the communication mechanism between different parallel tasks associated with two neighboring stream nodes. We also assume any transformation node maintains a sequential processing semantics. This means that a node processes each received record in a FIFO fashion instead of withholding certain records for a delayed emission.

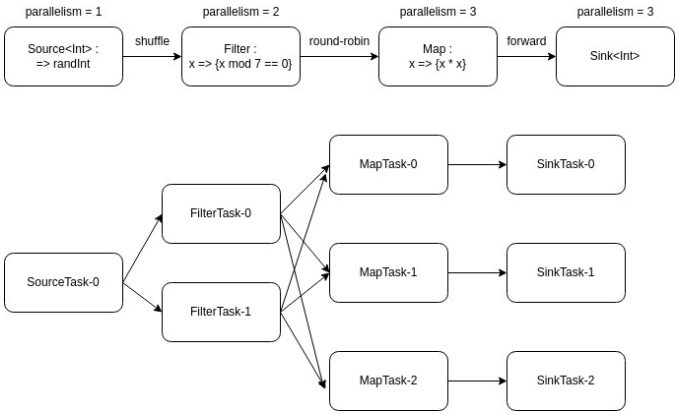

Fig. 2. Example stream graph (top) and the corresponding execution graph (bottom).

B. Execution Graph

An execution graph extends the logical representation of a stream graph to a physical execution scheme. Each stream node is mapped into a set of replicated tasks based on its assigned parallelism. To illustrate such transformation, we show in Fig. 2 a simple stream graph where a source node generates a stream of random integers. The integers are first filtered to keep only multiples of 7 and then further squared in the map transformation and eventually sent to the sink node. Given their corresponding parallelism and stream edge definition, we arrive at the execution graph in Fig. 2 where each node represents a parallel task and each arrow represents a communication channel. In our problem setting, the pipeline contains exactly one source task, which generates a single sequence of data that defines the ordering that Conflux seeks to preserve. We also assume such communication channels preserve FIFO ordering as is observed in most stream processing frameworks that use TCP as the underlying network protocol.

C. Formal Problem Definition

In this paper, we limit the scope of target user pipelines to the ones with stream graphs containing exactly one source node, one sink node, and an arbitrary number of transformation nodes. Both the source and sink nodes will have parallelism equal to one to generate exactly one input and one output data sequence. The transformation nodes can have arbitrary parallelism but will be constrained to have

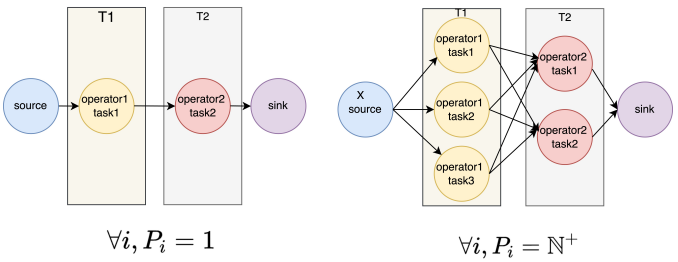

Fig. 3. An example of a parallel pipeline (right) and its sequential counterpart (left).

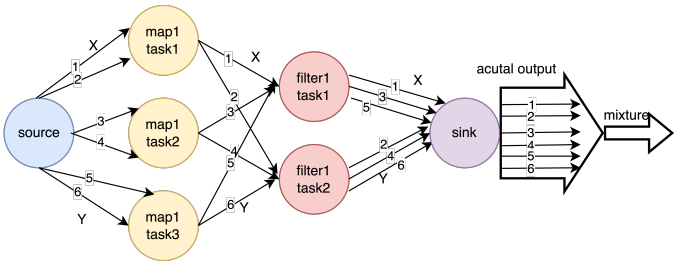

Fig. 4. Data paths in a sample pipeline

both stream edge in-degree and out-degree equal to one. This prevents order ambiguity resulting from stream merges and cycles which we will leave as a part of the future work.

Given a stream graph that complies with the above constraints, we define each stream node to be a tuple of the form $Nx(P, D)$ where Nx is the node ID, P is the parallelism and D is its output dispatch pattern to its downstream neighbor. A stream graph that satisfies our preliminary is thus a list of stream nodes

$$N_0(P_0 = 1, D_0), \ldots, Ni(P_i, D_i), ..., N_x(P_x = 1, null)$$

where N_0 is the source node and has parallelism equal to one. The last node Nx is the sink node and therefore has a dispatch pattern D_x equal to null. Any node $N_i(0 < i < x)$ is a transformation node that reads the output stream from N_{i-1} and sends data to node N_{i+1}.

We define the k-th parallel task of node $N_i(P_i, D_i)$ to be $N_i[k](0 \leq k < P_i)$. A dispatch pattern is thus a function that, for any $0 \leq i < x, 0 \leq k < P_i$, maps $N_i[k]$ to the subset of all possible connection pairs $\{(N_i[k], N_{i+1}[j])|\forall 0 \leq j < P_{i+1}\}$. A data record may be dispatched through any number of connection links in the set.

Finally, we define a pipeline with stream graph

$$N_0(P_0 = 1, D_0), \ldots, Ni(P_i, D_i), ..., N_x(P_x = 1, null)$$

to be source-order-preserving if its output sequence respect the output ordering of its strictly sequential counterpart where $P_i = 1 \quad \forall i \quad s.t \quad 0 \leq i < x$.

Fig.3 shows an example of a parallel pipeline and its sequential equivalent. This means that if record R_i is emitted before R_j in the sequential pipeline, the same must be the case for the corresponding parallel pipeline. In the case where records are duplicated across connection links, the output sequence of the parallel pipeline becomes partially ordered where there is no order requirement between duplicated records.

IV. Conflux Design

In this section, we explain the major components of Conflux and our implementation on top of Apache Flink and Apache Kafka.

A. Observation & Key Ideas

Our approach leverages the observation that records within a data path can maintain their relative ordering under certain conditions. Specifically, if both operator and communication channels retain FIFO ordering, then the relative ordering is preserved within a data path. The data path originates from a data source, travels through the job graph by traversing one parallel task per transformation, and arrives at a data sink. As illustrated in Fig. 4, arrows labeled with the same number compose a data path; for instance, record X follows path 1. Hence, when both the parallel instances of the operators and the communication channels uphold FIFO semantics, input records passing through the same data path retain their relative order. By capitalizing on the sorted nature of each data path, we transform a global sorting challenge into a k-way merge problem, which substantially reduces sorting overhead.

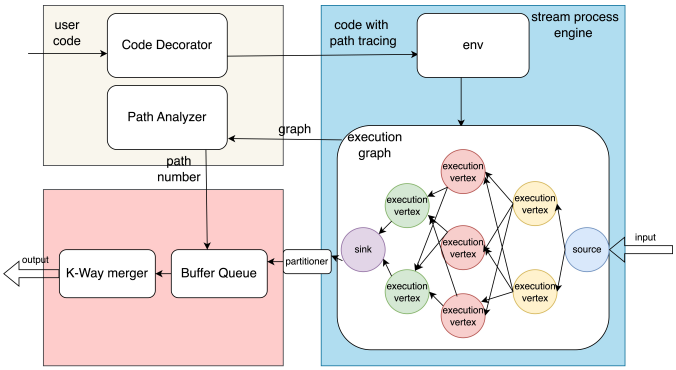

Fig. 5. Architecture overview of Conflux design.

However, one issue is that the records received at a sink come from various data paths, as demonstrated in Fig. 4. Consequently, it becomes necessary to track the path traveled by each record and utilize it to ensure that all records traversing the same data path are emitted to the same queue, forming new streams that are ready for k-way merge.

B. Workflow Overview

For the sake of simplicity, we do not incorporate a sequence number assigner at the source but rather assume that the source generates records with incrementing sequence numbers later used for sorting. As illustrated in Fig 5, the high-level workflow of Conflux is as follows: First, the Code Decorator transforms a user pipeline into a Conflux pipeline by injecting path-tracing code into each transformation operator and replacing the sink node with Conflux Sink. Next, the Path Analyzer examines the execution graph produced by the stream processing framework to generate a static mapping between each data path and buffer queue. Based on this static analysis, Conflux creates the required number of buffer queues and configures the partitioning strategy for the Conflux Sink. Finally, Conflux launches the downstream k-way Merger before executing the stream processing pipeline. We will explain each component and its implementation in the following sections and the full Conflux implementation can be found at https://github.com/Robertation256/flink-path-tracker.

C. Code Decorator

The Code Decorator transforms a user pipeline by allowing path tracing and attaching a watermark disseminating partition-aware Conflux Sink. To track the data path of a record, a tracer component is injected into each operator. The tracer simply requests the globally unique task ID from the framework runtime environment and appends it to the reserved path field in the record. This path eventually takes the form of

$$N_0[k_0], \ldots, N_i[k_i], ..., N_{x-1}[k_{x-1}]$$

where N_{x-1} is the last transformation before the sink in the user pipeline and $Ni[k_i]$ is the ID of the k-th parallel instance of transformation N_i.

Next, the Code Decorate replaces the user sink with the Conflux Sink which is configured with the same parallelism as the last user transformation node and receives records in a forward dispatch pattern (one-to-one communication link between instances of the last user transformation node and the Conflux sink instances). The Conflux Sink dispatches received records to the corresponding buffer queue based on the record's path field and the static mapping configuration later provided by the Path Analyzer. Moreover, the Conflux Sink extracts received watermarks emitted by the framework and broadcasts them as dummy records to all buffer queues corresponding to the data paths that reaches the sink instance.

Due to time limit, both modifications are currently manually applied to the user pipeline but we believe our design is relatively easy to achieve with annotated user code and simple program translation.

D. Path Analyzer

To ensure the correctness of the k-way merge, we need to compute the precise number of data paths given a user pipeline specification. One solution is to directly compute based on the semantics of the dispatch patterns observed in the pipeline. However, this is unnecessarily complicated given the rich collection of built-in and user-defined dispatch patterns. Therefore, the Conflux Path Analyzer adopts a simpler, yet more generic approach by traversing the execution graph generated by the framework engine. This ensures reliable path computation resistant to potential changes induced by framework-specific optimization.

Given a set of data paths explored from the execution graph, Conflux statically establishes a one-to-one mapping between data paths and buffer queues. This mapping is then used to configure the Conflux Sink and to create the buffer queues, for which we adopt Apache Kafka and use topic partitions as buffer queues.

E. Watermark Manipulation

Watermark is a common strategy for signaling pipeline progress in a stream processing setting [11]. In the case of Conflux, watermarks are configured to be periodically emitted at the source and carry the current largest logical timestamp (sequence number) seen at the source. As mentioned above, watermark signals propagated to the Conflux Sink are broadcasted to the buffer queues and serve as dummy records that prevent empty queues from halting the K-Way Merge process. As a result of the watermark semantics and its framework implementation in Flink, a dummy watermark record with sequence number X never precedes an actual record with sequence number X in the buffer queue.

F. K-Way Merger

The K-Way Merge Algorithm refers to a family of algorithms that merge K sorted lists into a single sorted sequence. In the case of Conflux, we employ a min heap implementation that allows merging unbounded data streams from the K unique data paths in a user pipeline. Algorithm 1 outlines the pseudocode for our K-Way Merge implementation. Heap entries are tuples containing both the record and its queue ID. Comparison is based on the record sequence number with the exception that a dummy watermark record is always considered larger than a regular record when their sequence numbers are equal. This ensures that regular records are promptly flushed out whenever possible. During operation, the K-Way Merger is first initialized by pushing one element from each queue onto the heap. When an entry is popped from the heap, the Merger polls the next record from the queue it belongs to and blocks when the queue is temporarily empty.

In our implementation over Apache Flink and Apache Kafka, the K-Way Merger employs K threads to fetch records from the remote Kafka partitions that serve as our buffer queues. Moreover, since Flink emits a final watermark with logical timestamp equals to `Long.MAX_VALUE` when the pipeline finishes, the Merger quits refilling for a queue upon observing this termination watermark.

Algorithm 1 K-Way Merger

Variables:

- *heapEntry*: A tuple containing a record and the index of the buffer queue it originates from
- *heap*: The min heap containing heapEntry
- *queues*: An array of merger-local queues populated from remote buffer queues.

```
1:  for idx ← 1 to queues.size do
2:      record ← queues[idx].blockingPoll()
3:      heap.push(HeapEntry(record, idx))
4:  end for
5:  while heap is not empty do
6:      heapEntry ← heap.pop()
7:      qId ← heapEntry.queueIdx
8:      record ← heapEntry.record
9:      seqNum ← record.sequenceNumber
10:     if record is not a watermark then
11:         emit(record)
12:     end if
13:     if seqNum ≠ Long.MAX_VALUE then
14:         next ← queues[qId].blockingPoll()
15:         heap.push(HeapEntry(next, qId))
16:     end if
17: end while
```

G. Discussion

In this subsection, we discuss and review the implications of the Conflux design.

Component Decoupling: Conflux design naturally allows the decoupling of record buffers from the stream processing pipeline where storing records in stream node local states can be prohibitively costly due to periodic system checkpoints. Moreover, since such sorted queues can be repeatedly consumed by multiple downstream services, applications may choose to deploy their own K-Way Merger variants. Applications less sensitive to order inversion may place a timeout on the blocking queue poll operation to trade order guarantee for higher

throughput.

Computation Overhead: A major benefit that comes with Conflux is that it allows stream nodes to scale without developers worrying about disrupted record ordering. Beyond that, when compared with topology-agnostic sorting implementations, Conflux enjoys a significantly smaller complexity overhead by exploiting the FIFO nature of each data path. Since the number of data paths K is a constant, Conflux imposes a minimal linear overhead.

Space Overhead: While additional space overhead is not unique to Conflux as window sort and global sort also require buffers, it is a reasonable concern that the amount of queue required by Conflux may increase quickly as parallelism increases. Nevertheless, by avoiding unnecessary data re-shuffling between stream nodes and frequent changes of parallelisms between neighboring nodes, the number of data paths can be significantly reduced. For example, given transformation node $N_i(P_i, D_i)$ and the immediate downstream node $N_{i+1}(P_{i+1}, D_{i+1})$, D_i can be a simple forward dispatch pattern that maps $N_i[k]$ to the connection pair $(N_i[k], N_{i+1}[k])$ when $P_i = P_{i+1}$. In this case, data path cardinality will not increase. Moreover, adhering to the forwarding pattern wherever possible is a general principle in stream processing pipeline development for avoiding latency introduced by network shuffling and leaving room for optimizations such as operator chaining [12].

Finally, we believe that Conflux's queue component comes at a very low hardware cost as each queue can be implemented as a log file. Mainstream solutions such as Apache Kafka recommends 4000 partitions (which are used as individual queues in Conflux) per broker and 200k partitions per cluster [13], which should accommodate fairly large-scale streaming pipelines.

Fault Tolerance and Consistency: Our implementation of Conflux enjoys the fault tolerance mechanisms and different levels of consistency guarantees provided by Apache Flink and Apache Kafka. In terms of fault tolerance, Flink offers global snapshots through checkpoints and Kafka provides redundancy through partition duplication. Regarding consistency guarantees, Conflux easily achieves at-least-once semantics by leveraging Flink checkpoints and Kafka offset commit. To ensure exactly-once semantics, however, user will need to activate both Flink's Two-Phase Commit and Kafka Transaction. Therefore, in general, Conflux does not address any fault tolerance and consistency requirements but rather, inherits such good properties from sophisticated frameworks it depends on.

Other Caveats: Nevertheless, certain caveats are intrinsic to the Conflux design. First, the watermark emission rate must be carefully configured so as not to stall K-Way Merger progress when a data path becomes vacant for a long period. Moreover, Conflux currently only handles relatively simple stream graph topology. We believe that an interesting area of future work would be providing a framework for users to define custom ordering semantics that handle more complicated cases such as stream joins and cycles.

V. Evaluation

In this section, we discuss our evaluation setup and assess Conflux's performance against a window-based baseline pipeline. We also study our evaluation results to answer the following questions:

- How does Conflux's end-to-end latency compare to the baseline? (§V-E)
- How does Conflux's throughput compare to the baseline? (§V-F)
- What is the additional latency overhead introduced by Conflux when compared with the barebone user pipeline? (§V-G)

A. Test Environment Setup

We deployed our test infrastructure over 20 VMs provided by the CS VM Farm at the University of Illinois Urbana-Champaign. Each VM runs on Red Hat Enterprise Linux 8 (64-bit), 2 vCPUs, 4GB RAM and, 100 GB storage. As shown in Fig. 6, an Apache Hadoop YARN cluster is deployed over VM1 to VM16 with VM1 reserved for centeral coordination services including HDFS Namenode, YARN Resource Manager, and Zookeeper. A Flink job is then submitted at VM1 and scheduled onto VM2 to VM16. A live Flink pipeline emits records to the Kafka Cluster deployed on VM18 to VM20. The K-Way Merger runs on VM17 and continuous polls and merges records from Kafka. The merged

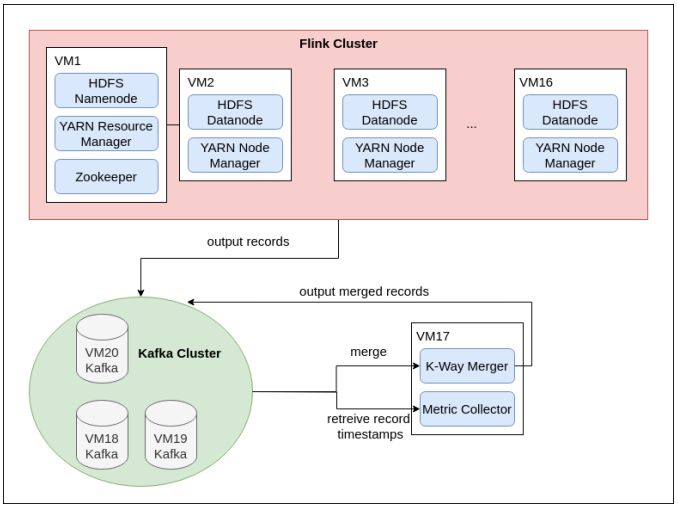

Fig. 6. Test infrastructure overview

```
datasource
    .filter(new
       TestRichFilterFunctionImpl())
    .setParallelism(15)
    .forward()
    .map(new TestRichMapFunction())
    .setParallelism(15)
    .rebalance()
    .map(new TestRichMapFunction())
    .setParallelism(2)
    .forward()
    .sinkTo(new DiscardSink<>());
```

Listing 1. Test pipeline definition.

record stream is then emitted back to the Kafka cluster under a separate topic. Upon pipeline completion, the metric collector on VM17 is manually triggered to collect and extract timestamps from all records emitted by the K-Way Merger. The same workflow applies to the baseline pipeline except that the K-Way Merger is not involved.

B. Test Pipeline Workload

Timestamp	Description
CreationTs	Time when a record is created at the source
ProcessTs	Time when the user pipeline finishes processing a record
SinkTs	Time when a record reaches the Sink Operator
ConsumeTs	Time when a record is fetched from buffer queues to the K-Way Merger
EmitTs	Time when a record is emitted by the K-Way Merger

TABLE I
RECORD TIMESTAMPS

To have fine-grained control over input records, we implemented a test data source with configurable input size and record payload size. The records emitted from the source carry strictly increasing sequence numbers and several timestamp fields that are populated as records pass through major progress checkpoints in the system. The timestamps are used solely for measuring system performance and the details can be found in Table. I.

Listing. 1 shows the test pipeline definition. The above-mentioned data source is first filtered to remove records with sequence numbers that are multiple of 7. The stream is then further processed by two dummy identity map operators. Both the filter and the two map operations are injected with a configurable number of while loops as additional workload.

C. Baseline Pipeline

We implemented a sorting pipeline based on Flink's TumblingWindow as our baseline. A sorter operator with parallelism equal to 1 is inserted between the last user transformation and the sink. The baseline pipeline adopts the same watermark strategy as seen in Conflux, which uses the record sequence number as logical timestamp to ensure correctness. When a window is triggered at the sorter, all records within the window are sorted using Java's built-in `Collections::sort`. The records are then passed onto the sink and then emitted to the Kafka cluster for storage. A sequence number checker is attached at the end of both the baseline and the Conflux pipeline to check for output order inversion.

D. Metrics

Here we explain several metrics we used to evaluate the performance:

1) End to End latency: The end-to-end latency measures the time interval between the record creation time at the source and the final emission time. This metric is crucial for assessing the efficiency of the processing pipeline, as it encompasses all

processing stages, and can be used to measure both baseline and Conflux fairly.

2) Source to Sink Latency: Source-to-sink latency measures the total time taken for a data record to travel from the source to the sink. For the baseline pipeline, this metric is equivalent to end-to-end latency. However, in the case of Conflux, this metric specifically measures the process time of the user pipeline as no sorting is performed before the sink. By comparing it with the end-to-end latency, we can determine the additional latency introduced by Conflux.

3) Source to Merger Latency: This latency measures the time it takes for records to travel from the source to the local queues at the K-Way Merger. This metric is collected for the Conflux pipeline, accounting for the latency introduced by the buffer queues.

4) Throughput: The throughput is defined as the number of records outputted by the pipeline over a 3-second interval.

5) Pipeline Completion Time: The pipeline completion time measures the time between the moment the first record enters the pipeline and when the last record is emitted. This metric offers a direct and intuitive indication of the overall processing efficiency of a stream processing system.

E. How does Conflux's End-to-end Latency Compare to the Baseline?

In this section, we evaluate the end-to-end latency of Conflux and the baseline. We use the pipeline described in V-B with 10 million input records and 1000-cycle while loop workload to measure the end-to-end latency.

As demonstrated in Fig. 7, Conflux demonstrates superior performance in terms of end-to-end latency. The CDF graph indicates that approximately 80% of the records in the Conflux pipeline are processed and emitted within 1500 milliseconds, whereas the baseline pipeline only manages to process the same percentage of records within approximately 2250 milliseconds.

More specifically, as shown in Fig. 8, Conflux exhibits a mean end-to-end latency of 1330.20 milliseconds, which is significantly lower than the baseline's mean latency of 2097.33 milliseconds, representing a reduction of approximately 36%. Conflux achieves a P99 latency of 2074.00 milliseconds compared to the baseline's higher 2915.00

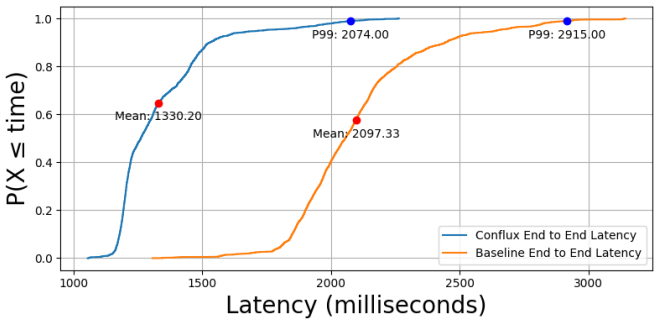

Fig. 7. CDF of end-to-end latency when running our test pipeline with 10 million records

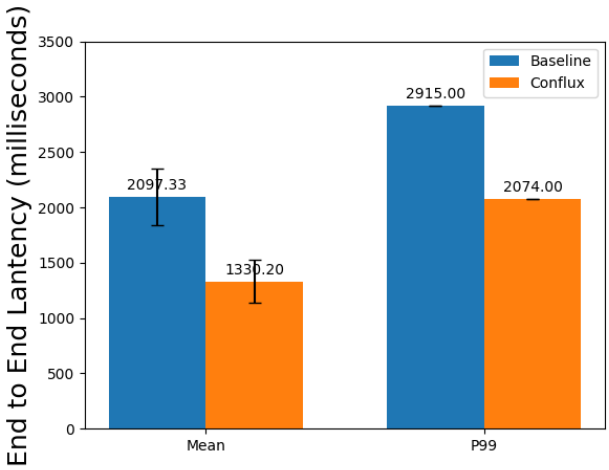

Fig. 8. Mean and P99 end-to-end latency of Conflux and baseline pipeline (both with 10 million input records).

milliseconds. The reason of the performance difference is likely due to the fact that the baseline solely depends on watermark for driving system process whereas in Conflux, records can be safely emitted when all data paths are populated.

F. How does Conflux's Throughput Compare to the Baseline?

In this section, we measure the throughput of per-second intervals with a 3-second slide window. The experiments are performed with 10 million input records.

As depicted in Fig. 9, the Conflux pipeline starts with a sharp increase, ultimately stabilizing at a throughput exceeding 250,000 records per second. In contrast, the baseline pipeline achieves a steady

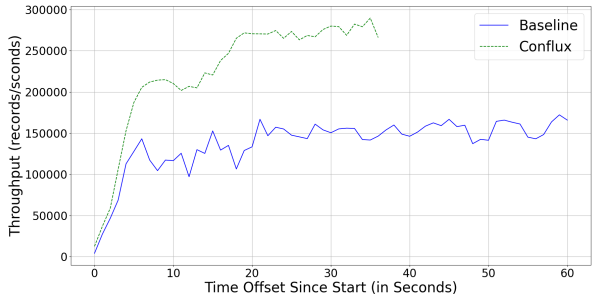

Fig. 9. Conflux and baseline throughput. The x-axis represents the time offset since the start of the pipeline in seconds.

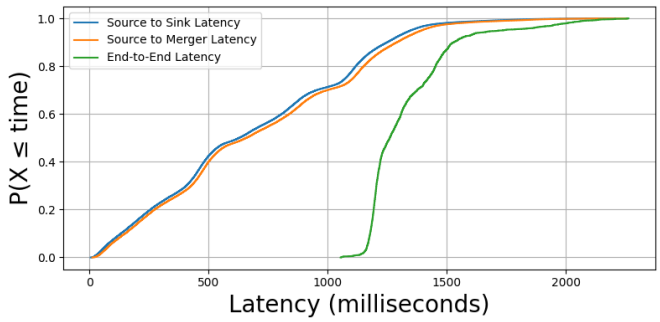

Fig. 10. CDF of various latency's for Conflux with 10 million input records.

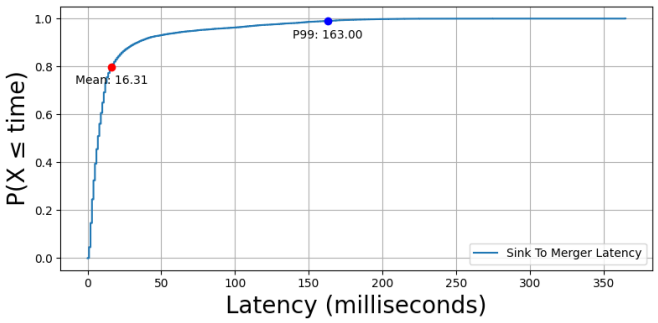

Fig. 11. CDF of sink-to-merger latency

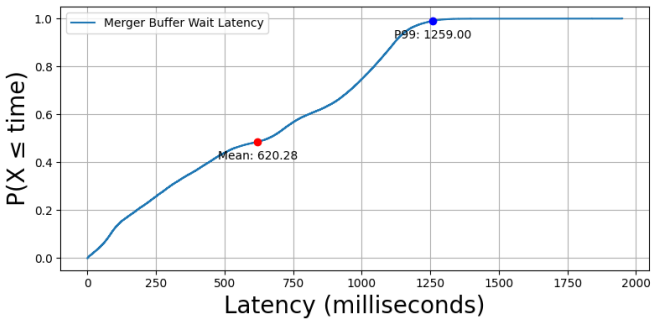

Fig. 12. CDF of merger-local queues to emission latency, i.e., the time interval between when a record is retrieved by the merger to when it is emitted.

throughput ranging between 140,000 and 150,000 records per second, which represents a reduction in throughput of more than 40% compared to the Conflux pipeline. Notably, across all measured time intervals, Conflux consistently outperforms the baseline in terms of throughput efficiency.

Additionally, in Fig. 9, we observe that Conflux finishes processing 10 millions records after 36 seconds. In comparison, the baseline pipeline takes 60 seconds to complete processing the same amount of records.

G. What is the Additional Latency Introduced by Conflux?

Now, we demonstrate a latency breakdown of each of the components of the Conflux pipeline. Fig. 10 illustrates the cumulative distribution (CDF) of latency in milliseconds for source-to-sink latency, source-to-merger latency, and end-to-end latency.

The curves representing source-to-sink latency and source-to-merger latency in Fig. 10 are nearly aligned, suggesting that the path from sink to the merger contributes only a small additional delay. Specifically, as illustrated in Fig. 11, the buffer queue introduces an average latency of approximately 16.31 milliseconds, with a 99th percentile latency of 163 milliseconds.

Our analysis of the latency incurred by different Conflux components indicates that the primary delay rises from the K-Way Merger. As demonstrated in Fig. 12, the latency from when a record enters the merger to when it is emitted exhibits a mean of 620.28 milliseconds and reaches a P99 of 1259.00 milliseconds.

VI. Conclusion

This paper is inspired by a real-world engineering dilemma where a trading compliance application requires its upstream parser pipeline to retain event order in its output while keeping up with massive market data volume. We present Conflux, a novel technique for efficiently preserving source

stream order in the context of a distributed stream processing system through k-way merge by leveraging knowledge of the pipeline topology and the FIFO property of data paths. Our implementation of Conflux over Apache Flink and Apache Kafka significantly outperforms the baseline, achieving approximately 41% lower P99 end-to-end latency and 40% higher average throughput.

Important future work includes but is not limited to providing a semantic framework for working with more complicated user pipelines that contain join operations or multiple source streams, as well as integrating Conflux as part of the framework support in mainstream processing engines such as Spark Structured Streaming and Apache Flink.

REFERENCES

[1] Windows. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/dev/datastream/operators/windows/#windows

[2] Structured streaming programming guide. Apache Spark. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[3] Monitoring back pressure. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/ops/monitoring/back_pressure/#back-pressure

[4] State backends. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/ops/state/state_backends/

[5] Checkpointing. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/

[6] M. Jeon and D. Kim, "Parallelizing merge sort onto distributed memory parallel computers," in *Proceedings of the 4th International Symposium on High Performance Computing*, ser. ISHPC '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 25–34.

[7] Y. C. Kim, M. Jeon, D. Kim, and A. Sohn, "Communication-efficient bitonic sort on a distributed memory parallel computer," in *Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001*, 2001, pp. 165–170.

[8] M. H. Nodine and J. S. Vitter, "Deterministic distribution sort in shared and distributed memory multiprocessors," in *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 120–129. [Online]. Available: https://doi.org/10.1145/165231.165247

[9] M. Hofmann and G. Runger, "A partitioning algorithm for parallel sorting on distributed memory systems," in *2011 IEEE International Conference on High Performance Computing and Communications*, 2011, pp. 402–411.

[10] Z. Khatami, S. Hong, J. Lee, S. Depner, H. Chafi, J. Ramanujam, and H. Kaiser, "A load-balanced parallel and distributed sorting algorithm implemented with pgx.d," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 1317–1324.

[11] Generating watermarks. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/event-time/generating_watermarks/

[12] Task chaining and resource groups. Apache Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/overview/#task-chaining-and-resource-groups

[13] How to choose the number of topics/partitions in a kafka cluster? Confluent. [Online]. Available: https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/