Adaptive Handling of Out-of-order Streams in Conformance Checking

Kristo Raun[1,*], Riccardo Tommasini[1,2] and Ahmed Awad[3]

[1]University of Tartu, Tartu, Estonia
[2]LIRIS Lab, INSA de Lyon, France
[3]The British University in Dubai, Dubai, United Arab Emirates

Abstract

Organizations function through the execution of various business processes. Non-conformant behavior in these processes impacts organizations negatively through implications such as reduced efficiency, lower quality, and compliance risks. Thus, it is important to identify non-conformant process behavior rapidly. While this is a challenging problem on its own, it is further complicated by the advent of big data, distributed systems, and a fragmented landscape of cloud- and on-premise tools that all provide data for the analysis of a business process and for determining its conformance. In such a landscape, it is common that events may arrive out of order. This complicates the conformance-checking analysis, which commonly expects events within a process to arrive in a specific sequence allowed by the process model.

This paper introduces the first streaming conformance-checking method that incorporates event time awareness, thus having the ability to correct imperfections stemming from the out-of-order arrival of events. The method is scalable, utilizing the Beamline framework built on top of Apache Flink. Furthermore, the method includes an adaptive approach for handling various levels of out-of-order events in event streams. Experiments were conducted to demonstrate the applicability of the method for real-world use cases with different levels of out-of-order events. The results indicate that the method is well suited for identifying conformance in business processes that rely on a multitude of underlying systems for aggregating a holistic view of the process.

Keywords

Streaming conformance checking, Real-time analytics, Process mining, Scalability

1. Introduction

Business processes are at the core of all organizations. Understanding, measuring, and monitoring business processes helps organizations become more efficient, have higher quality of work, and have higher compliance. Over the last two decades, *process mining* has emerged as the field that studies business processes to improve process executions through the systematic use of event data [1].

An event log contains the event data of a process, while a process model describes the allowed behavior within a process. Example process model and event log are depicted in Figure 1 and Table 1, respectively. The execution of a process instance may not necessarily align with the constraints set by the process model. *Conformance checking* [2] in process mining refers to the comparison of the expected behavior, depicted by a process model, and the actual behavior, represented as event data. While traditional conformance checking is done in an offline setting, an array of research in recent years has focused on conducting conformance checking in a streaming setting [3, 4, 5].

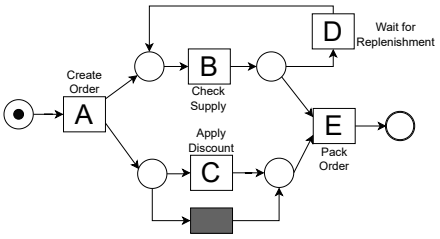


Figure 1: Example of a process model.

Streaming conformance checking shares many similarities with data stream processing: high volume and velocity of data, low latency requirements, unboundedness of data streams, and stream imperfections [6, 7]; however, while the former items have garnered attention in research in recent years, the area of stream imperfections has remained neglected. Handling out-of-order events has been investigated from the perspective of a process discovery setting [8], but there have been no known works on conformance checking with out-of-order event streams.

Event streams do not commonly exhibit a constant level of out-of-orderedness [9]. Rigidly dealing with out-

DOLAP 2024: 26th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2024, March 25, 2024, Paestum, Italy
*Corresponding author.
kristo.raun@ut.ee (K. Raun); riccardo.tommasini@liris.cnrs.fr (R. Tommasini); ahmed.awad@buid.ac.ae (A. Awad)
0000-0001-7535-2084 (K. Raun); 0000-0003-3404-5250 (R. Tommasini); 0000-0003-1879-1026 (A. Awad)

Case ID	Activity	Timestamp
1	A	2023-12-01 09:00
1	B	2023-12-01 09:01
2	A	2023-12-01 09:03
1	D	2023-12-01 09:03
3	A	2023-12-01 09:04
2	C	2023-12-01 09:04
1	B	2023-12-01 09:07

Table 1
Example of an event log.

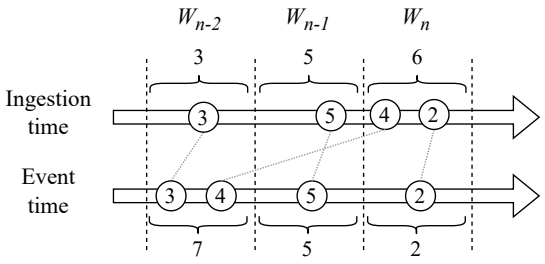


Figure 2: Example of out-of-order event arrival.

of-order data may overcompensate at times when events arrive mostly on time and undercompensate when out-of-order event arrival is frequent. Thus, ideally, any method that handles out-of-order event arrival should be adaptive in responding to the stream's characteristics. Such an adaptive streaming conformance checker would be useful in industries where it is critical to have an up-to-date overview of process executions, but the underlying systems may introduce out-of-order events; for example, healthcare, clickstream analytics, manufacturing, and smart mobility.

The contributions of this paper are threefold: (1) a streaming conformance-checking approach capable of handling out-of-order events, (2) a novel approach for adaptive handling of event-time progress in business process streams, and (3) lifting an existing state-of-the-art approach to using Apache Flink, allowing for a truly streaming and scalable conformance checking solution.

The rest of the paper is structured as follows: in Section 2, we introduce the background, discussing out-of-order event arrival in data streams, business process characteristics, and the current state of the art in streaming conformance checking. In Section 3, we discuss an existing conformance-checking to be extended to handle out-of-order event streams. We look at how the method would handle out-of-order events and how the solution can adapt based on the frequency of out-of-order event arrival. Section 4 describes the setting for running the experiments and the results of the experiments, together with a discussion of the results. Finally, Section 5 concludes the work.

2. Background

2.1. Out-of-order Event Streams

Modern stream processing frameworks are designed for analytics that build upon recent data stream aggregates, especially in cases where real-time results are essential, such as systems monitoring and decision support [6, 10]. A data stream that features data arriving from multiple sources or a system built upon distributed systems will likely display a stream imperfection known as *out-of-order* event arrival [11]. An event is considered out of order when it arrives from a source after records with later event timestamps. Many factors can explain why a stream may have events arriving out of order, the most typical reasons being network reliability, bandwidth, and load [12].

Because a data stream is inherently infinite, the memory requirements on a stream processing system would be unbounded. Thus, the concept of windows is used to segment time into smaller units [13]. Windows allow aggregations and joins of multiple streams within specific timeframes. In the presence of out-of-order events, windows need to be maintained and possibly entirely recalculated. An example of out-of-order event arrival in the context of windows, and a comparison between event time and system ingestion time is shown in Figure 2. In this example, aggregating based on windows would indicate an increasing trend if looking at ingestion time $(3, 5, 6)$, while based on the event time, it is actually a downward trend $(7, 5, 2)$.

Since there is no limit on the potential delay of event arrival, an additional concept called *watermarks* is used in data streams to keep track of event time progress and, essentially, limit the number of windows [14]. This avoids potential time lag due to outliers or faulty data that could stretch windows indefinitely. While commonly, watermarks are based on event time [12], this is complicated for business process streams due to the characteristics of business processes and the fact that commonly, a single process execution does not have direct dependencies to other concurrent process executions, as we will see next.

2.2. Business Process Characteristics

Data analytics answers questions such as what happened and when [15]. However, traditional data analytics does not generally consider data from a business process viewpoint. Processes have a defined course of action and constitute specific activities [1], thus enabling answers to questions such as why something happened.

Process models describe the behavior allowed in a process. Many notations exist with various characteristics,

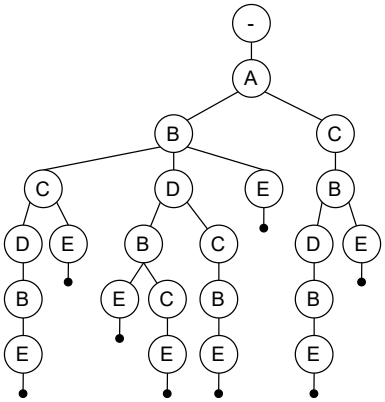


Figure 3: A trie representing process behavior.

such as Petri Nets, Directly-Follows Graphs, and BPMN models [1]. Most process models visualize the activities within the process and allow for behavior such as sequences, parallelism, choices, and loops. In the context of this paper, the method employs the trie data structure for describing process behavior [16]. A shortcoming of the trie is that it cannot explain parallelism and loops, thus being an approximation by a sample of the full process behavior; however, the trie is well suited for conformance-checking purposes, where the goal is to compare event data to the allowed behavior efficiently. An example of the trie sampled from the process model in Figure 1 is shown in Figure 3.

The actual process executions are stored in event data. Commonly, a single event contains attributes such as the *case identifier (Case ID)*, *activity*, and *timestamp*. All activities having the same case identifier make up a *trace* – a specific process execution instance. Events can be stored in static logs, called *event logs*, or in *event streams*, where each event arrives as it occurs in the real world.

2.3. Streaming Conformance Checking

Conformance checking is the examination of process executions and quantifying the non-conformant behavior compared to the behavior allowed by a process model [2]. The state-of-the-art output of a conformance checker is an alignment between the actual behavior (log moves) and a complete run allowed by the model (model moves) [17]. Alignments have the benefit of being more readily interpretable than most other diagnostic methods and have thus enjoyed wide adoption by the research community. An example alignment is shown in Table 2, with the log moves $\langle A, D, E\rangle$ indicating the actual process executions and the model moves $\langle A, B, D\rangle$ indicating a valid path in the model. The skip symbol $\gg$ indicates non-conformant behavior, typically associated with a conformance cost.

Traditional conformance checking is done on event logs extracted from business systems. While such a static approach is robust, it has some shortcomings; most notably, the data is obsolete by design. Thus, several recent research efforts have made conformance checking on event streams viable [3, 18, 5, 4].

log moves	A	$\gg$	D	E
model moves	A	B	D	$\gg$

Table 2
An example (prefix-)alignment for the trace $\langle A, D, E\rangle$.

The work in [4] introduced the concept of alignments to streaming conformance checking by utilizing prefix alignments. In event streams, it is unknown whether the process executions have terminated, and a prefix alignment has the benefit of not penalizing the observations by expecting an entire model run. At the same time, a prefix alignment in a streaming setting allows a high level of interpretability of the discrepancy, which becomes especially relevant in larger processes.

While the initial prefix-alignment-based approaches offered guarantees on the exactness of the results, they suffered from a relatively long latency, rendering the applicability of the methods challenging for faster event streams. Thus, new approaches have emerged recently. In [5], an approximate approach was introduced – *IWS* – for calculating prefix alignments on top of event streams. The algorithm utilizes the trie data structure as the allowed process behavior, improving the calculation time of alignments in some cases by several orders of magnitude while introducing only a moderate error compared to the previous state of the art. The work in [18] built upon the IWS algorithm, extending the approach with completeness and confidence metrics, allowing for quantification of warm-starting scenarios and a trace's potential conclusion in a stream. It is the first prefix-alignment-based method that supports warm-starting scenarios in streaming settings – i.e., cases where the stream started before the conformance checker began measuring the results. In the next section, we will look deeper into the IWS approach to see why we think it is well suited to handle stream imperfections, and we present extensions to the original algorithm to handle out-of-order events and to adapt to changing frequency of out-of-order event arrival.

3. Methodology

3.1. Decay Time: Tracking Event Progress in IWS

The IWS algorithm [5] allows for fast and approximate conformance checking while under memory constraints in a streaming setting. For efficiency, IWS uses the trie

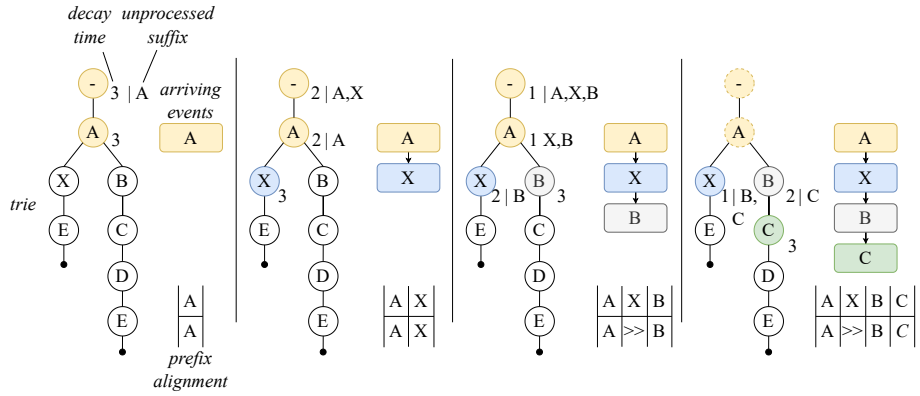


Figure 4: State buffer evolution with arriving of events. The color coding visually links the arriving event with the state and its node calculated.

data structure to compare the event data to the expected process behavior. Due to the nature of business process executions, it may sometimes be necessary to re-calculate the optimal route in the trie, and due to erroneous activities, the method may end up on a *wrong* path in the trie. Thus, the IWS approach uses a *state buffer* for keeping some past states available for recalculation and a *discounted decay time* hyperparameter to release old states from memory. The decay time is calculated based on the difference between the average length from the root to a leaf node in the trie, indicating a roughly average expected trace length, and the current trace length. The outcome is multiplied with the *discounting factor*, a value between 0 and 1 to indicate the maximum percentage of the average trace length that should be kept in memory. For a more detailed explanation and formula, we refer to [5]. Conceptually, the decay time helps the algorithm progress event time on a per-trace basis. Traces may overlap and have different event execution patterns. Thus, progressing based on event time, as it is commonly done with watermarks, would favor traces with rapid executions, while time-consuming process instances would be quickly forgotten, and the analysis would suffer. Thus, the decay time setting in IWS is not time-based but event-based in the scope of a specific trace.

An example of the state buffer and the decay time are shown in Figure 4. With the arrival of the first event, A, two states are initiated, with the state at the root node holding A in its unprocessed suffix. The unprocessed suffix is used to replay the moves upon the arrival of the next events. Decay time indicates how many events within this trace should arrive before the state will be cleared from memory. The optimal alignment at each event arrival is also shown. For event arrival B, there are actually multiple optimal alignments, but only one alignment is shown in the figure for illustrative purposes. Importantly, the state buffer allows the retraction of the false path traversal to node X. This behavior can be repurposed to implement out-of-order handling into the algorithm, as will be described in the next section.

3.2. Event Time Store

To handle out-of-order events, the algorithm is extended by an event time store that keeps the event time of each arrived event. The event time store is an ordered key-value pair, with the key denoting the event time and the value being an array of events that occurred during this time. For simplicity, we assume that if events have the same event time, the arrival order is the correct total order of these events. In other words, the array denotes the arrival time of the events having this event time. Formally, assume that $\mathcal{T}$ is the set of timestamps, $\mathcal{E}$ is the set of events, and $\mathcal{E}*$ is the set of all possible words over $\mathcal{E}$, then the event store $\mathcal{S}$ is a function $\mathcal{S} : \mathcal{T} \rightarrow \mathcal{E}*$. As the decay time releases states from memory, the event time store releases the earliest events from the event store.

For out-of-order handling, each event time is first compared to the largest key in the event store as events arrive. If the new event has a timestamp equal to or larger than the largest key, this event is arriving in order, and processing continues as usual. If the largest key is larger than the timestamp of the arrived event, all the events with a larger timestamp in the event store are considered out-of-order events and are piped for a new replay. Furthermore, any states in the state buffer that have played out any out-of-order events are removed from the buffer, while the rest of the states remove the unprocessed suffix that matches the new sequence of events.

To illustrate, let's consider an example of allowed behavior as shown by the trie in Figure 3.

Assume we observed events $\langle A, D, E \rangle$ with event timestamps of 1, 3 and 5, respectively. While this trace could have multiple optimal alignments, for simplicity, assume we have the same alignment as in Table 2.

If the algorithm now receives event B with an event

Event time aware	A	B	D	E	
	A	B	D	≫	
Non-event time aware	A	≫	D	E	B
	A	B	D	≫	B

Table 3
Comparison of event time aware and non-aware alignments.

timestamp 2, it first checks the event store to see whether the events are arriving in order. The event store's largest key (5) is larger than the arrived event timestamp (2). Thus, all events with timestamps larger than 2 are considered out-of-order, and all of the states in memory that have played out the events D and E are removed. The resulting alignment of the event time aware solution is shown in Table 3, with a comparison to the original non-event time aware version that assumes all events arrive in order, leading to a higher conformance cost.

3.3. Adaptive Event Time Progress

The arrival of out-of-order events cannot be expected to be static throughout the life of the stream. Thus, approaches in stream processing have been devised to adapt the watermarks based on concept drifts – changes in data arrival frequency and delays [9]. This paper introduces a novel approach for adaptive event time progress suited for business process data. Namely, we adopt the Exponentially Weighted Moving Averages (EWMA) metric from inventory and financial planning [19] and modify it to work as a sensor for indicating the level of out-of-orderedness.

To adapt to the stream's frequency of out-of-order events, we extend the algorithm with the following method to modify the discounting factor. With every new event, we check if the event is out of order. If it is out of order, we assign it a boolean value of 1 and 0 if it is not. Then, we increase (or decrease) the discounting factor using the following formula:

$$df = \alpha * ooo + (1 - \alpha) * df$$

Where df is the discounting factor, α is the smoothing factor, and ooo is the Boolean value of whether it is an out-of-order event. In our experiments, we found an alpha of 0.005 to represent an appropriate change in the discounting factor.

Intuitively, if the proportion of out-of-order events has increased, then the discounting factor will increase, thus keeping in memory a larger amount of states and allowing for improved out-of-order event handling. If the frequency of out-of-order events decreases, so too will the discounting factor, releasing the memory strain. We consider it unlikely that any specific trace would start exhibiting out-of-order behavior while other traces would have events in order. Thus, the formula is applied globally to all traces within the process.

3.4. Implementation

In order to be truly scalable, the original algorithm and the extensions introduced in this paper are implemented on top of the Beamline framework [20]. The Beamline framework utilizes Apache Flink as the runtime engine, allowing the algorithm's execution to scale across a cluster of computing nodes. Commonly, each individual trace in a business process is looked at separately. Thus, partitioning by the case identifier would theoretically allow scaling of the processing to as many nodes as there are cases within the process.

The source code for the implementation, together with instructions for running the experiments and the datasets used, have been made available on GitHub[1].

4. Experiments

4.1. Setting

Several real-life event logs were used to test the handling of out-of-order events. The logs had to be manipulated to mimic the out-of-order scenario, as the original logs were grouped by trace and in temporal order. The logs used in this paper are well-known real-life process event logs: BPI 2012[2], BPI 2017[3], and BPI 2020 Travel Permits[4].

The steps done for running the experiments are shown in Figure 5. To limit the scope of the experiments, the logs were first randomly sampled to 100 traces (step 1). Then, events within a trace were swapped with various settings ranging from no out-of-order events to fully out-of-order events (step 2). The settings are described in Table 4, showing the probability of a swap, i.e., how likely a single event is to trade places with another event within the same trace, and max distance, i.e., how far from the current position can an event be swapped to. For example, with the *swap_01* setting, each event has a one percent likelihood of getting swapped with a maximum distance of one, meaning that it will trade places with the event directly before or after.

The unaltered sampled log was used for generating the trie – the process model that describes the expected behavior (step 3). Since the IWS algorithm is capable of streaming conformance checking, the experiments were also conducted in a streaming fashion using an MQTT broker. A Python script published the out-of-order logs to MQTT topics (step 4), and the algorithm

[1] https://github.com/DataSystemsGroupUT/StreamingConformanceChecker
[2] https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
[3] https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
[4] https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51

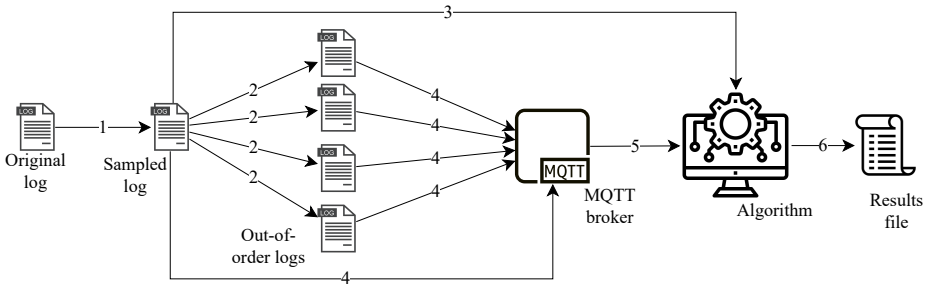


Figure 5: Experiment settings

setting name	probability of a swap	max distance of a swap
swap_00	0	0
swap_01	0.01	1
swap_05	0.05	5
swap_10	0.1	7
swap_20	0.2	10
swap_50	0.5	20
swap_99	0.99	99

Table 4
Swap settings.

received the events by subscribing to these topics (step 5). The results of the experiments were outputted to a file on an event-by-event basis (step 6), measuring the latency of the algorithm – how long it takes to process an event – and the cost of the latest alignment of the case to where this particular event belongs to.

The method was instantiated with the default settings for the *decay time* variable: a minimum decay time (dt) of 3 and a discounting factor (df) of 0.3. The method was run with the adaptive add-on from Section 3.3 turned on (*adaptive*), turned off (*non-adaptive*), and the out-of-order handling turned off (*non-aware*, i.e., the original IWS algorithm).

All experiments were executed three times and averaged to mitigate possible runtime outliers impacting the results. The experiments were conducted on a machine using Java 11 and Python 3.9. The MQTT broker used was EMQX 5.1, which was run using Docker.

4.2. Results

4.2.1. Latency.

Figure 6 shows the latency, i.e., the processing time of the algorithm per event, in milliseconds. For the BPI2012 log, there is almost no difference on processing time for the various swap variations until $swap\_50$, where the out-of-order handling shows a clear penalty in terms of processing time. For the $swap\_99$ variation, the non-adaptive version is almost an order of magnitude slower than the non-aware IWS version. The adaptive method is a further order of magnitude slower than the non-adaptive method, with values of 1.89 ms/event for non-aware and 139.32 ms/event for the adaptive methods.

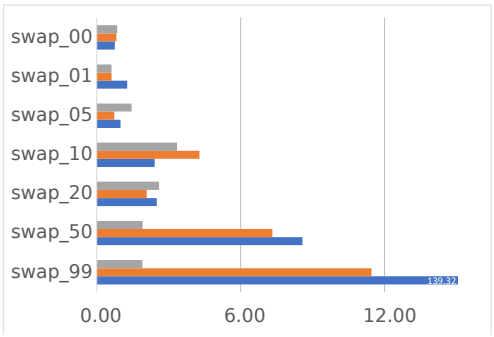
For the other datasets, a similar pattern can be observed. For BPI2017, the difference between non-aware and out-of-order handling methods is clear starting from $swap\_50$, and for $swap\_99$ the difference between non-aware and the adaptive method is almost three orders of magnitude. It can be observed that the execution time of the non-aware version of the algorithm does not increase with increased out-of-orderedness – this is because the algorithm does no recalculation for out-of-order event arrival and simply assumes that there is a great deal of non-conformant behavior occurring in the event stream.

For BPI2020 log, the results vary slightly more, with non-aware and non-adaptive versions being roughly equivalent for $swap\_50$, and the adaptive method is faster than the non-adaptive method for $swap\_99$. However, this may be due to the fact that this is the smallest of the datasets, as can be seen by the execution time remaining under 1-2ms per event.

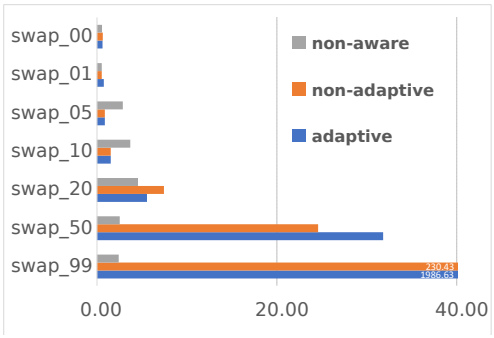
4.2.2. Cost.

A core measure of a conformance checker is the cost. In this paper, we measure the cost of an alignment, i.e., similarly to an edit distance difference between the expected process behavior and the actual observed behavior. The benefit of using an alignment is explainability, indicating clearly which part of the process contains the discrepancy. It is important to remember that the non-aware version naively assumes that the order in which the events arrive is the order in which the events happened, thus negatively impacting the alignment cost because the events were actually in the correct order but swapped. The cost results are summarized in Table 5, with color-coding from green (the best result) to red (the worst result) per log and swap variation.

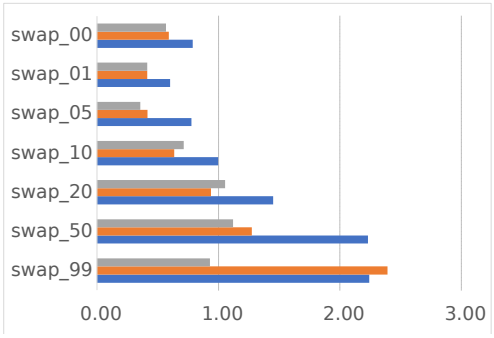
An observation can be made that as the amount of swaps increases, the cost increases. This is true for all executions, except the adaptive algorithm on BPI2012 that slightly decreases cost for $swap\_99$ compared to



(a) BPI 2012



(b) BPI 2017



(c) BPI 2020

Figure 6: Latency per event in milliseconds.

$swap\_50$. This is due to the randomness of the out-of-orderedness in the generated event data. In general, the cost increase is due to the fact that with the swaps, we have introduced superficial non-conformant behavior. As was shown in Table 4, the higher swap settings increase the likelihood and distance of an event displacement, thus having an increased amount of non-conformant behavior.

Comparing the different versions of the algorithm, it is clear that the non-aware version severely penalizes the out-of-order events. The difference between adaptive and non-adaptive versions is minuscule until $swap\_20$, when the adaptive versions starts to outperform the non-adaptive version, and the $swap\_50$ and $swap\_99$ variations have an almost double the difference in cost, in favor of the adaptive version.

4.3. Discussion

Based on the results, we can say that we have introduced a streaming conformance-checking approach capable of handling out-of-order events. Furthermore, it seems that the adaptive handling of event-time progress is well suited for adapting to an increased load of out-of-order event arrivals. In general, the introduced methodology seems to work well for handling out-of-order events, even for streams where the portion of out-of-orderedness is relatively high. As expected, higher amounts of out-of-order events impact latency negatively. At the same time, the cost is greatly improved compared to the original IWS algorithm, which is unaware of event time. For smaller business processes, such as BPI2020, the adaptive method has low latency even with extreme out-of-orderedness. However, with more complex business processes, such as BPI2017, the non-adaptive method may be more sensical from the latency perspective.

Some threats of validation include the fact that only a few datasets were used in this comparison. Furthermore, the out-of-orderedness had to be mimicked because no known process mining logs or streams that exhibit out-of-order events are publicly available. The swap settings could be further investigated, as the probabilities and max distances could increase orthogonally, not in correlation.

One thing to address in future research would be the fact that if multiple events have the same timestamp, then the method should not blindly assume that the arrival order within the timestamp is correct. This is seen, for example, on the BPI2012 dataset, with many simultaneous timestamps. Having a method that would be able to find the optimal solution from partial order would be a further improvement to the introduced approach.

Ultimately, as the results are positive, and the latencies are generally low for most experiments, we believe this method would be applicable for real-life use cases for running conformance checking on distributed systems.

5. Conclusion

This paper introduced an approach for handling out-of-order event arrival in streaming conformance checking. To the best of our knowledge, this is the first conformance-checking approach that is aware of event time and, thus, is able to handle such stream imperfections. The contribution included a novel approach for adapting the event time progress based on the frequency of out-of-order event arrival. The approach was implemented using the Beamline framework built on top of

	BPI2012			BPI2017			BPI2020		
	adap.	non-adap.	non-aw.	adap.	non-adap.	non-aw.	adap.	non-adap.	non-aw.
swap_00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
swap_01	1.1	1.1	1.6	0.0	0.0	3.0	0.0	0.0	0.8
swap_05	1.3	1.3	6.0	1.9	1.9	18.7	1.4	1.4	5.0
swap_10	7.1	7.1	13.7	3.9	4.6	35.6	6.7	6.5	11.5
swap_20	12.1	13.2	24.9	5.3	12.6	66.0	8.8	10.5	18.4
swap_50	22.2	33.0	44.9	10.3	32.9	101.0	9.3	16.9	27.0
swap_99	21.9	38.6	48.7	24.5	44.0	108.4	9.9	20.7	28.2

Table 5
Cost comparison. Average alignment cost per trace.

Apache Flink, making the method easily scalable and able to handle large data volumes in stream processing.

Future research aims to look at handling partial order in event streams, as the current method assumes the arrival order for events having the same event time is the total order. Furthermore, a more extensive study could be done of the swap settings in event streams and their impact on out-of-order event handling.

Acknowledgments

This work was supported by the European Social Fund via "ICT programme" measure, the European Regional Development Fund, and the programme Mobilitas Pluss (2014-2020.4.01.16-0024).

References

[1] W. M. van der Aalst, Process mining: a 360 degree overview, in: Process Mining Handbook, Springer, 2022, pp. 3–34.

[2] J. Carmona, B. F. van Dongen, A. Solti, M. Weidlich, Conformance Checking - Relating Processes and Models, Springer, 2018.

[3] A. Burattin, J. Carmona, A framework for online conformance checking, in: International Conference on Business Process Management, Springer, 2017, pp. 165–177.

[4] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, W. M. van der Aalst, Online conformance checking: relating event streams to process models using prefix-alignments, International Journal of Data Science and Analytics 8 (2019) 269–284.

[5] K. Raun, R. Tommasini, A. Awad, I will survive: An event-driven conformance checking approach over process streams, in: DEBS, ACM, 2023, pp. 49–60.

[6] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, S. Khan, A survey of distributed data stream processing frameworks, IEEE Access 7 (2019) 154300–154316.

[7] A. Burattin, Streaming process mining, Process Mining Handbook (2022) 349–372.

[8] A. Awad, M. Weidlich, S. Sakr, Process mining over unordered event streams, in: 2020 2nd International Conference on Process Mining (ICPM), IEEE, 2020, pp. 81–88.

[9] A. Awad, J. Traub, S. Sakr, Adaptive watermarks: A concept drift-based approach for predicting event-time progress in data streams., in: EDBT, 2019, pp. 622–625.

[10] S. Zhang, J. Soto, V. Markl, A survey on transactional stream processing, The VLDB Journal (2023) 1–29.

[11] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, D. Maier, Out-of-order processing: a new architecture for high-performance stream systems, Proceedings of the VLDB Endowment 1 (2008) 274–288.

[12] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, arXiv preprint arXiv:2008.00842 (2020).

[13] K. Patroumpas, T. Sellis, Window specification over data streams, in: International Conference on Extending Database Technology, Springer, 2006, pp. 445–464.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, The Bulletin of the Technical Committee on Data Engineering 38 (2015).

[15] Z. Sun, L. Sun, K. Strang, Big data analytics services for enhancing business intelligence, Journal of Computer Information Systems 58 (2018) 162–169.

[16] A. Awad, K. Raun, M. Weidlich, Efficient approximate conformance checking using trie data structures, in: 2021 3rd International Conference on Process Mining (ICPM), IEEE, 2021, pp. 1–8.

[17] W. van der Aalst, A. Adriansyah, B. van Dongen, Replaying history on process models for conformance checking and performance analysis, WIREs Data Mining and Knowledge Discovery 2 (2012) 182–192.

[18] K. Raun, M. Nielsen, A. Burattin, A. Awad, C-3PA: Streaming conformance, confidence and completeness in prefix-alignments, in: International Conference on Advanced Information Systems Engineering, Springer, 2023, pp. 437–453.
[19] P. R. Winters, Forecasting sales by exponentially weighted moving averages, Management science 6 (1960) 324–342.
[20] A. Burattin, Streaming process mining with beamline, ICPM Demos (2022).