Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams

Ahmed Awad
University of Tartu, Estonia
ahmed.awad@ut.ee

Jonas Traub
Technische Universität Berlin
jonas.traub@tu-berlin.de

Sherif Sakr
University of Tartu, Estonia
sherif.sakr@ut.ee

ABSTRACT

Event-time based stream processing is concerned with analyzing data with respect to its generation time. In most of the cases, data gets delayed during its journey from the source(s) to the stream processing engine. This is known as *late* data arrival. Among the different approaches for out-of-order stream processing, low watermarks are proposed to inject *special* records within data streams, i.e., watermarks. A watermark is a timestamp which indicates that no data with a timestamp older than the watermark should be observed later on. Any element as such is considered a late arrival. Watermark generation is usually *periodic* and heuristic-based. The limitation of such watermark generation strategy is its rigidness regarding the frequency of data arrival as well as the delay that data may encounter. In this paper, we propose an adaptive watermark generation strategy. Our strategy decides adaptively when to generate watermarks and with what timestamp without a priori adjustment. We treat changes in data arrival frequency and changes in delays as concept drifts in stream data mining. We use an Adaptive Window (ADWIN) as our concept drift sensor for the change in the distribution of arrival rate and delay. We have implemented our approach on top of Apache Flink. We compare our approach with periodic watermark generation using two real-life data sets. Our results show that adaptive watermarks achieve a lower average latency by triggering windows earlier and a lower rate of dropped elements by delaying watermarks when out-of-order data is expected.

1 INTRODUCTION

Stream analytics is concerned with analyzing data on the move. Several stream processing engines (SPEs) have been developed that vary in their processing capabilities. One fundamental feature is the ability to process data with respect to their generation time rather than their arrival time at the SPE [2]. This is commonly known as event-time versus processing-time. For a data stream element e, we define $te(e)$ as a function that returns the timestamp of the event, i.e. the time it was created at the source, we also define $tp(e)$ to be the timestamp when it was first seen by the SPE. The event-time notion is more relevant in several applications. However, as the data sources are distributed and can be placed far away from SPEs, *delays* can occur until an element e reaches the SPE. This is known as *late arrival*. Moreover, a stream element can arrive *out-of-order*. For example, two stream elements e_1 and e_2 where e_1 was created before e_2, i.e., $te(e_1) < te(e_2)$ can arrive in the opposite order, i.e., $tp(e_2) < tp(e_1)$. This is known as out-of-order arrival.

Stream analytics jobs are usually defined by a sort of a window [4, 5, 7]. Windows are used to slice the infinite stream into finite chunks and apply the analytics computation, e.g. averaging, a.k.a window function, on the content of the window. Time windows are common types of windows that define stream element's membership to the window based on the elements timestamp w.r.t. window start and end times, when working in event time. The window function cannot be applied on the window content until the SPE decides that the window is closed. For system (processing) time mode, the decision is made when the internal clock (wall clock) of the SPE passes the window end. However, for event time processing, we need an external notion progress.

Low watermarks generation [1, 2] is a technique to account for the progress in event time. A watermark is a timestamp that *indicates* that there should be no future data older than the watermark to be seen. When a watermark is received by a window operator in a pipeline, it triggers executions of completed time windows whose closure completed at a time earlier than the watermark. A watermark is monotonic. That is, a new watermark cannot be less than the last generated watermark. Watermark progress affects two metrics in stream processing: latency and late arrival of data items and thus accuracy. Latency is hurt when too few watermarks are generated. Late arrival increases when too many watermarks are generated and thus accuracy decreases due to more elements being ignored (dropped).

Currently, watermark generation is either heuristic-based and periodic [1, 2] or punctuation-based [11]. The latter assumes knowledge about the content of the data. The former is inflexible with the change of the distribution of data arrival rate or the distribution of the delays in arrival. Those are unique properties of data streams. In this paper, we contribute an approach to watermark generation that is data-content-agnostic. We call our approach as adaptive watermarks. By treating the change in the skewness between stream elements event time and processing time as a concept drift, we employ a drift detection technique, the adaptive window (ADWIN) [3, 6], to decide when to generate a new watermark and with which value. Moreover, we provide another control, late arrival threshold, to decide about the watermark generation. We implement our approach on Apache Flink and compare it with a baseline periodic watermark generator using two real-life data sets. Our experiments empirically prove the superiority of adaptive watermarks with respect to a reduced latency and/or a reduced number of dropped elements.

2 ADAPTING TO DATA ARRIVAL RATES

2.1 Baseline: Periodic Watermark Generation

Periodic watermark generation is a heuristic-based approach. Heuristically, application developers determine a max allowed lateness m. With the arrival of a new event, the *maximum* timestamp seen is updated. Then, every s milliseconds, the SPE obtains the maximum timestamp t seen so far. The watermark value is $t - m$. New stream elements that arrive after the watermark are considered late. Depending on the configuration of the stream processing pipeline, such elements can still be included in the

Table 1: Parameters used for watermark generators

Parameter	Description
δ^*	Sensitivity to change $\in [0, 1]$. Default is 1.
l	Late arrival threshold, $l \in (0, 1]$. Default is 1.
m^*	Skewness between event time and ingestion time.
Δ_δ	Sensitivity change ratio, $\Delta_\delta \in (0, 1]$. Default is 1.
w	Warmup tuples used to initialize m.

Parameters with * are derived by the system

result or ignored. This approach is simple to implement. However, it is does not adapt to changes in 1) the arrival rate of data elements, 2) the lateness of elements.

2.2 Adaptive Watermark Generation

In practice, it would be desirable to learn both m, the lateness, and the period s for generating a new watermark and keep updating them as new elements arrive. Moreover, to account for high correctness, one would hold the generation of new watermarks if the ratio of late arrival elements to the total elements goes above a certain threshold l since the last generated watermark. In this situation, the period s at which a watermark is generated will change (adapt) according to the change in the distribution of data inter-arrival time. Similarly, m can be learned upon each change detection. The change of the inter-arrival time can be seen as a concept drift in data streams [10]. In our case, the drift to be detected is the change of events' inter-arrival time. To detect this drift, we employ the ADWIN algorithm [3].

ADWIN is an algorithm which detects concept drifts (e.g., changes in users opinions) and enables adapting machine learning models on data streams. It works by maintaining a window of data items over time. The size of the window changes over time based on the frequency of change detected in incoming data. The algorithm does not require a pre-determined period to trigger change detection. It checks for drifts on a per-tuple basis. The more change occurs, the smaller the size of the window as older items are considered irrelevant and dropped. The algorithm has a single parameter δ (Table 1) which controls the sensitivity to change. The higher the value of δ the more sensitive it will be to change. In our work, δ is by default set to 1 so that a change is detected as early as possible.

ADWIN works as follows: Upon the arrival of a tuple, it is added to the window. Then, the algorithm tries to detect a change by finding two sub-windows whose value distributions are *significantly different* with respect to δ. Here, ADWIN iterates over all possible combinations of sub-windows. As an optimization, ADWIN maintains histograms (buckets) of the sum and variance of the data rather than the data itself to have a better memory footprint. The histogram grows logarithmically to the number of data points. Grulich et al. [6] parallelize different parts of the original algorithm and reach two orders of magnitude enhancement of its throughput. Table 1 summarizes the parameters we use for the adaptive watermark generation technique.

Algorithm 1 describes how we employ change detection by ADWIN to adapt the generation of watermarks. At the arrival of a new element, that might be late, the *difference* between the new element's timestamp and its ingestion timestamp is inserted into ADWIN and a check is made for detection of a change (drift) (Line 12). Such a detection can be an indicator to generate a new watermark. It is not necessary that every change detection leads to the generation of a new watermark. We need to look at the second indicator which is the rate of late arrivals. A new

Algorithm 1: Adaptive watermark generation

```
   Input: A data stream S
   Input: Sensitivity change rate Δ_δ
   Input: Late arrival threshold l
   Input: Warmup tuples count w
1  warmup ← 0; m ← 0; watermark ← 0;
2  lateElements ← 0; totalElements ← 0; δ ← 1
3  maxTimestamp = −∞
4  adWin ← initializeAdWin(δ)
5  foreach e ∈ S do
6      maxTimestamp = max(te(e), maxTimestamp)
7      if warmup ≤ w then
8          m ← max(m, tp(e) − te(e))
9          warmup ← warmup + 1
10     else
11         totalElements ← totalElements + 1
12         if adWin.driftDetected((tp(e) − te(e))/m, δ)
            then
13             if lateElements = 0 then
14                 δ = increaseSensitivity(Δ_δ)
15             if lateElements/totalElements < l then
16                 watermark = maxTimestamp − m
17                 emit(watermark)
18                 lateElements ← 0
19                 totalElements ← 0
20             else
21                 m ← updateSkewness()
22                 δ = decreaseSensitivity(Δ_δ)
23         else
24             if te(e) < watermark then
25                 lateElements ← lateElements + 1
```

watermark is generated only if this rate is less than the threshold l at the time of change detection (Line 15). This rate is reset each time a new watermark is generated.

We call the elements collected between two successive watermarks a *chunk*. The value of m is the maximum difference between elements' ingestion and event time within the chunk (Line 21). The value of the new watermark is *maximum timestamp*$-$ m (Line 16). Upon the generation of a new watermark, all data about late arrival is reset (Lines 18 and 19). In case a chunk has no late arrivals, the sensitivity is increased (Line 14) to speedup change detection. However, upon crossing the late arrival threshold l, δ is adapted according to the δ_Δ parameter (Line 22).

As ADWIN is designed to work with values in the range $[0, 1]$, we need to normalize the deviation between element's event and ingestion times. For this, we use the first w tuples of the stream to learn about m (Line 7). That can be in the form of minimum, maximum, or average skewness observed between elements' event time and ingestion time.

For the adaptive watermark generation, the user needs to specify the late arrival threshold l, the sensitivity change ratio δ_Δ, and the warmup tuples w. The default values for these parameters are 1, 1, and 10000 tuples, respectively. The choice of a value of l is mainly driven by the accuracy expected by the application. The more accurate results expected, the lower the value of l should

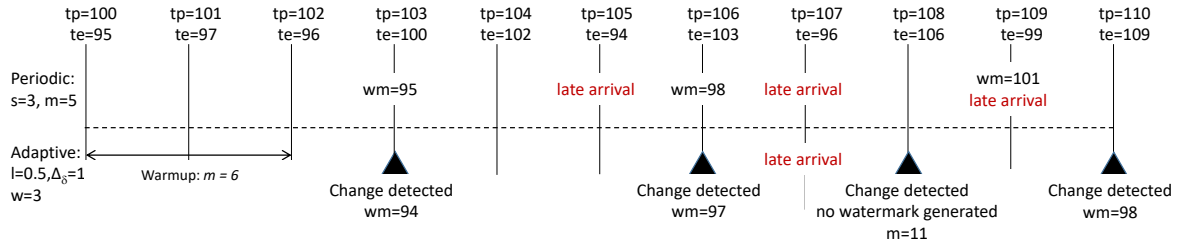

Figure 1: Example for adaptive watermark generation versus periodic generation.

be. If the source is stable and does not produce considerable late arrivals, the low value of l should not affect the latency of the results. Otherwise, lower values of l shall affect the latency, but this should be acceptable as the main concern is accuracy. Sensitivity to change δ_Δ is effective only in case that late arrivals rate is above l. In such case, δ is decreased and thus ADWIN will have to observe a larger number of elements before detecting a change. The value of w affects latency as for the first w tuples, no watermarks are generated and thus no window contents is processed. Larger value of w could be used with data sources with out-of-order arrivals. In Algorithm 1, we use the *maximum* skewness observed to update m. However, this can be changed to *average* or other measures.

Example: Figure 1 exemplifies how adaptive watermark generation works in comparison to the periodic one on a hypothetical stream. Vertical lines represent stream elements with their processing and event time respectively. The periodic generator is configured with $s = 3$ seconds and $m = 5$ seconds. The adaptive generator is configured with $l = 0.5$, $\Delta_\delta = 1$ and $w = 3$. The periodic generator will generate watermarks at $tp = 103$, $tp = 106$, and $tp = 109$ with values 95, 98, and 101, respectively. The elements arriving at $tp = 105$, $tp = 107$ and $tp = 109$ are late. For the adaptive generator, the first three tuples are used to initialize m; which is set to 6, as the max skewness observed between tp and te. At $tp = 103$, ADWIN detects a change. As there are no late arrivals, a watermark is generated with the value 94. At time $tp = 106$, another change is detected and a new watermark is generated with value 97. The next element at $tp = 107$ is late. ADWIN detects another change at time $tp = 108$. At this time, no watermark is generated as the late arrival ratio is equal to 0.5. Also, at this time, m is updated to 11, the skewness between $tp = 107$ and $te = 96$. At $tp = 110$, ADWIN detects another change. A watermark is generated as the late arrival ratio drops below 0.5.

3 EVALUATION

Setup: We have implemented adaptive watermark generation on top of Apache Flink v1.6.2, using the `Source` APIs to control the emission of watermarks. Our code base for the implementation and experimental evaluation can be found online[1]. We run our experiments on a standalone cluster with 6 *GB* of main memory and 4 cores at 2 *GHz*.

Data and query: For the comparison between adaptive and periodic watermark generation, we use two data sets from the *DEBS* grand challenges of 2012 [8] and 2015 [9] respectively. The *DEBS 2012* data set has 32,390,519 tuples and 1.5% of the elements arrive out-of-order with an average of 100 tuples per second. The *DEBS 2015* data set has 14,776,616 tuples and has 78.6% of its tuples arrive out-of-order. Moreover, the arrival rate of the data varies along the data set with an average of 6 tuples per second. For both of the data sets, we project the timestamps and create events with dummy content and the timestamps projected. We use a simple pipeline that applies a count function on the content of a time window. We report the window boundary, start and end, the number of elements in the window and the difference between window end and the watermark. The resulting tuples are of the form $\langle start, end, count, delay \rangle$. We use these values to construct the comparison metrics as we show next.

Table 2: Metrics for periodic watermark generation

Data-set	Allowed lateness	Period	Win. size	Dropped %	Avg. win. delay (ms)
DEBS 2012	1000	200	1000	1.24	9,452,454
			100	1.24	3,083,109
	100	10	1000	1.50	713,846
			100	1.50	175,497
DEBS 2015	1000	200	1000	98.72	644,046,759
			100	98.62	648,821,195
	100	10	1000	99.93	546,682,126
			100	99.97	392,904,397

Metrics: We measure two metrics: the percentage of dropped elements and the average delay between a window end and the watermark after which the window function was triggered. We call it the window delay and report it in milliseconds.

Parameters: For the periodic watermark generator, we set the value for the generation period $s \in \{10, 200\}$ *ms*. For the allowed lateness, $m \in \{100, 1000\}$ *ms*. For the adaptive watermark generator, we set w the number of tuples to initialize m to 10000, we set the late arrival threshold l and the sensitivity change rate Δ_δ to one of $\{1.0, 0.1, 0.01\}$, see Table 1 for descriptions of the parameters. For the size of the time window in the query, we vary it between 100 and 1000 milliseconds.

Results: For the data collected after running the pipeline for each unique combination of configuration parameters, to compute results we: 1) drop out result tuples for which the watermark is `Long.MAX_VALUE`. Flink generates a watermark with this value to flush any windows that were waiting for a watermark to trigger its computation, 2) count the number of tuples (*count* above) and subtract that from the total number of tuples we have in the respective data set to obtain the tuple drop percentage, and 3) average the delay over the computed windows remaining after step 1. Table 2 shows the results for the periodic generator and Table 3 shows the results for the adaptive one.

For the *DEBS 2012* data, the periodic generator provides lower delay for the shorter watermark generation period. But, with

[1] https://github.com/DataSystemsGroupUT/Adaptive-Watermarks

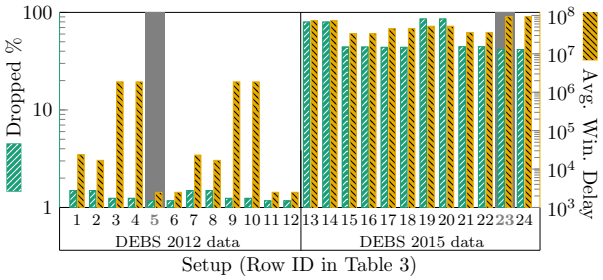

Figure 2: Evaluation of adaptive watermark generation

Table 3: Metrics for adaptive watermark generation

	Dataset	Δ_δ	l	Win. size	Dropped %	Avg. win. delay (ms)
1	DEBS	1	1	1000	1.49	23,633
2	2012	1	1	100	1.49	16,796
3		1	0.1	1000	1.24	1,853,667
4		1	0.1	100	1.24	1,847,622
5		1	0.01	1000	1.17	2,467
6		1	0.01	100	1.17	2,469
7		0.1	1	1000	1.49	22,905
8		0.1	1	100	1.49	16,796
9		0.1	0.1	1000	1.24	1,851,629
10		0.1	0.1	100	1.24	1,847,523
11		0.1	0.01	1000	1.17	2,472
12		0.1	0.01	100	1.17	2,471
13	DEBS	1	1	1000	79.60	72,863,504
14	2015	1	1	100	79.60	72,863,574
15		1	0.1	1000	44.14	33,826,576
16		1	0.1	100	44.14	33,827,281
17		1	0.01	1000	43.82	45,279,068
18		1	0.01	100	43.82	45,279,328
19		0.1	1	1000	85.52	51,965,257
20		0.1	1	100	85.52	51,965,876
21		0.1	0.1	1000	44.42	36,007,475
22		0.1	0.1	100	44.42	36,007,727
23		0.1	0.01	1000	41.64	92,672,393
24		0.1	0.01	100	41.64	92,673,050

higher drop rate. This is logical because of the trade off between tuple drop percentage and window delay. In the case of the adaptive watermark generator, with the default configuration (rows 1&2 in Table 3), it has almost the same tuple drop percentage as the periodic generator but with at least an order of magnitude improvement in window delay. Setting late arrival threshold $l = 0.1$, the tuple drop percentage is decreased but with two orders of magnitude higher window delay (rows 3&4). Yet, the delay is still below the delay of the periodic generator with the same tuple drop percentage. Setting $l = 0.01$ improves the tuple drop percentage by 0.07% and with very low window delay (row 5). This might look counterintuitive. But, by investigating the data for this configuration, we found that several windows were just fired by the termination of Flink's job. Thus, we had fewer windows that were processed due to the generated watermarks. This is logical as with a lower value of l, less number of watermarks is generated in general. Table 3 shows that changing the value of Δ_δ for the *DEBS 2012* data does not have much of an effect and the results are almost identical. The control is mainly driven by l.

For the *DEBS 2015* data set, we can see that the periodic generator drops almost all the tuples for the different configurations (Table 2). The window delay is very huge. Using the adaptive generator, with the default configuration (rows 13 and 14 in Table 3), we achieve a drop rate close to the percentage of the out-of-order tuples in the data set. However, the delay is an order of magnitude less than the periodic generator. Restricting the late arrival threshold to 0.1 reduces the tuple drop percentage and reduces the window delay by about 50% (rows 15 and 16). Pushing l to 0.01 reduces the tuple drop rate slightly but with an increase in window delay. The least tuple drop percentage is achieved when $\Delta_\delta = 0.1$ and $l = 0.01$ but with higher window delay (row 23). Figure 2 visualizes the tuple drop percentage and the average window delay for the different parameter values of the adaptive generator.

Our experimental results show the superiority of adaptive watermarks to baseline periodic ones. For both ordered and unordered streams, less tuple drop rate as well as several orders of magnitude saving in window delay.

4 CONCLUSION

In this paper, we have proposed an adaptive approach for generating low watermarks for event-time stream processing. It adapts to changes in data arrival frequency as well as to late arrival ratio. Compared to the baseline heuristic periodic watermark generation, and as indicated by application on data sets with different arrival frequency and delay characteristics, our approach strikes a balance between latency and accuracy of stream applications.

Currently, setting the late arrival threshold to very low percentage ceases the generation of the next watermark until the threshold is reached. This might be inconvenient and might cause high-latency. We intend to investigate an automated approach that automatically balances between latency and accuracy without having the user to specify threshold explicitly.

Acknowledgment: The work of Ahmed Awad and Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75). The work of Jonas Traub is funded by the German Ministry for Education and Research as BBDC II (01IS18025A).

REFERENCES

[1] Tyler Akidau et al. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB* 6, 11 (2013), 1033–1044.
[2] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803.
[3] Albert Bifet and Ricard Gavaldà. 2007. Learning from Time-Changing Data with Adaptive Windowing. In *SIAM*. 443–448.
[4] Nihal Dindar, Nesime Tatbul, Renée J Miller, Laura M Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *VLDB Journal* 22, 4 (2013).
[5] Buğra Gedik. 2013. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience* 44, 9 (2013), 1105–1128.
[6] Philipp Marian Grulich, René Saitenmacher, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2018. Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing. In *EDBT*.
[7] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. 2018. Stream Processing Languages in the Big Data Era. *SIGMOD Record* 47, 2 (2018), 29.
[8] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *DEBS*.
[9] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 Grand Challenge. In *DEBS*.
[10] Imen Khamassi et al. 2018. Discussion and Review on Evolving Data Streams and Concept Drift Adapting. *Evolving Systems* 9, 1 (2018), 1–23. https://doi.org/10.1007/s12530-016-9168-2
[11] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on KDE* 15, 3 (2003), 555–568.