Toward Sliding Time Window of Low Watermark to Detect Delayed Stream Arrival

Xiaoqian Zhang[1] and Kun Ma[2(✉)]

[1] School of Information Science and Engineering, University of Jinan, Jinan 250022, China
965326116@qq.com

[2] Shandong Provincial Key Laboratory of Network Based Intelligent Computing, University of Jinan, Jinan 250022, China
ise_mak@ujn.edu.cn

Abstract. Some emergency events such as time interval between input streams, operator's misoperation, and network delay might cause stream processing system produce unbounded out-of-order data streams. Recent work on this issue focuses on explicit punctuation or heartbeats to handle faults and stragglers (outlier data). Most parallel and distributed models on stream processing, such as Google MillWheel and Apache Flink, require hot replication, logging, and upstream backup in an expensive manner. But these frameworks ignore straggler processing. Some latest frameworks such as Google MillWheel and Apache Flink only process disorder on an operator level, but only point-in-time and fixed window of low watermarks are discussed. Therefore, we propose a new sliding time window of low watermarks to detect delayed stream arrival. Contributions of our methods conclude as adaptive low watermarks, distinguishing stragglers from late data, and dynamic rectification of low watermark. The experiments show that our method is better in tolerating more late data to detect stragglers accurately.

Keywords: Stream processing · Watermark · Out-of-order data · Stragglers · Late data

1 Introduction

Fault tolerance, low latency, balancing correctness, and checkpoints are features of streaming processing system [9]. Data arrival time does not strictly reflect its event time. It is important to distinguish whether the late data is simply delayed (due to the network delay and unbalanced allocation of the system resources) on the wire or actually lost somewhere [1]. If the stragglers (also called outlier data or slow streams) cannot be processed timely, the correctness of the processing result will be affected. Besides, stragglers decrease the running speed and increase error probability. However, current streaming processing systems has

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 350, pp. 444–454, 2021.
https://doi.org/10.1007/978-3-030-67540-0_28

little measures to address this fault tolerance issue [14,18]. In our paper, we try to detect delayed stream arrival in stream processing.

Logging, hot replication [13] (where every node has two copies), and upstream backup [8,12] (where messages are replied when tasks failed) are common method in stream fault recovery. Discretized stream (D-Stream) is abstraction layer of Spark Streaming system, which allows checkpointing and parallel recovery [6] and recover from fault to tolerate stragglers [18]. The method relies on micro batches in the technical stack of Spark system. An out-of-order processing framework is proposed to prevent ordering constraints [11] by AT&T Labs. x Google MillWheel proposed a low watermark strategy to solve this problem for the incoming stream and checkpoints its progress at fine granularity as part of this fault tolerance [1]. Low watermark is a limit of timestamp of all processing data. Apache Flink proposed the window mechanism of low watermark [5] to advance low watermarks of MillWheel. They use maximum allowed delay to allow data delay over a period of time. But these methods cannot distinguish straggler and late data. Apache Storm uses Trident (the high-level transaction abstraction) to guarantee exactly-once semantics for record delivery to avoid out-of-order data [15]. It requires strict transaction ordering to operate, and is based on a continuous operator model that performs recovery by replication (several copies of each node) or upstream backup (nodes buffer sent messages and replay them to a new copy of a failed node). This will cause extra hardware cost and take a long time to recover due to the transaction framework. More importantly, neither replication nor upstream backup handles stragglers. A straggler must be treated as a failure or fault in upstream backup, while it will slow down both replicas with synchronization protocols [14].

We propose a new method, adaptive sliding time window of low watermark, to make late data, stragglers and normal data from streams different. The method can reduce the number of late data and improve the accuracy of result. Unlike Low Watermark point-in-time of MillWheel, adaptive sliding time window of low watermark is a window. Adaptive low watermark, distinguishing stragglers from late data, and automatic correction of low watermark are unique compared to other methods.

The chapters of this paper are distributed as follows. Section 2 describes the latest work on out-of-order stream, low watermark and improved adaptive watermark. Section 3 introduces how to distinguish late data, stragglers and normal data. Firstly, we expand the point-in-time of watermark to sliding time window of low watermark. Secondly, using the window to define late data, straggler and normal data. Finally, we propose the delayed stream arrival detection algorithm. The experimental result, displayed in Sect. 4, shows the advantage of our method. Section 5 outlines brief conclusions and future research.

2 Related Work

2.1 Low Watermark

It is different between the order of arrival of tuples and the order of source [16]. Network delay and delivery errors cause disorder. Stragglers (also known as slow data) are inevitable in large clusters. Therefore, it is necessary to detect delayed

stream arrival for fault recovery. There is a mechanism which allows windows to stay open in additional time to process out-of-order stream data.

Low watermark is a timestamp that limit the timestamp of future data [1]. The low watermark of MillWheel tracks all events waiting to be solved in the distributed system to distinguish two situations. First, there is no late data arrival if the low watermark advances past time t without the data arriving. Second, there is late data with out-of-order stream arrival, which can solve timing issues in real-time computing. Given a computation A, the oldest work in A is a timestamp relevant to the oldest uncompleted record in A. The recursive definition of low watermark of A based on data flow is as follows.

Low watermark of A = min (oldest work of A, low watermark of C: C outputs to A). A and C are computations.

If there are no input streams, the low watermark is equipped to oldest work values.

2.2 Adaptive Watermark

Adaptive Watermark is a new generation strategy that can adaptively decide when to generate a watermark and which timestamp without any prior adjustments [2]. ADWIN (Adaptive Window) is used to detect concept drifts [7]. When the ADWIN detects concept drift, the algorithm will calculate dropped rate (computed by lateElements/totalElements). If this ratio is smaller than the threshold l (late arrival threshold), a new watermark will generate. When a new watermark is generated, the dropped rate is resetted. Adaptive Watermark can adapt to changes in data arrival frequency and delayed arrival rate, which realizes the balance between latency and accuracy. However, some fixed parameters like l are setted empirically, which may cause high-latency when processing data. Apache Flink provides abstractions that allow the developers to assign their own timestamps (timestamp assigner) and emit their own watermarks (watermark generator) [5]. That can be individualized by developers. In Apache Flink, the two watermark generation methods are periodic and punctuated [3]. Programmer can realize their thought by overwriting the *extractTimestamp* function. Allowed lateness is also a particularly improved feature of Apache Flink in dealing with disorderly events [4]. By default, when a watermark passes through the window, late data will not be discarded. In an improved way, allowed lateness allows for a period of time within an acceptable range (also measured by event time) to wait for the arrival of data before a watermark exceeds end-of-window in order to process the data again.

3 Delayed Stream Arrival Detection

3.1 Sliding Time Window of Low Watermarks

We propose the sliding time window of low watermark to distinguish different data types and receive data. Figure 1 shows the sliding time window of

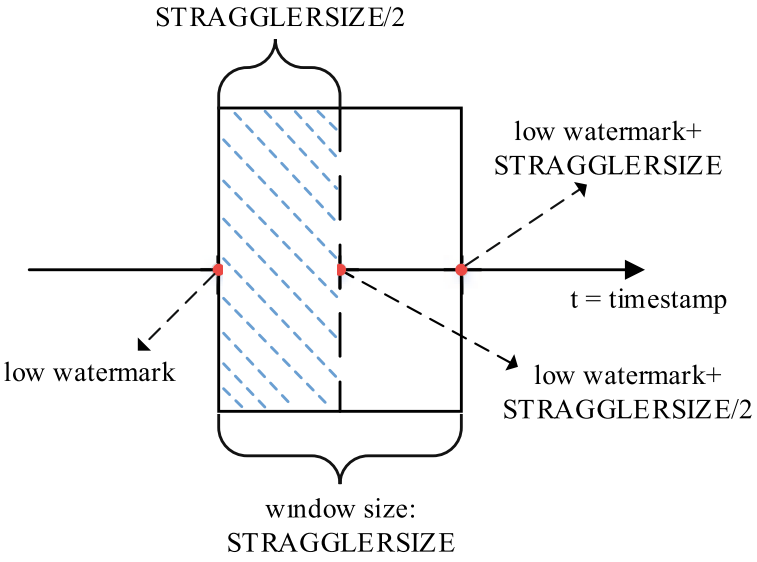

Fig. 1. Sliding time window of low watermark

low watermark in detail. As we can see from Fig. 1, the left of the window is *lowwatermark*, which represent a bound on the timestamps of future data. The size of window is $STRAGGLERSIZE$. The range of sliding time window is [$lowwatermark$, $lowwatermark + STRAGGLERSIZE$). $lowwatermark + STRAGGLERSIZE/2$, the middle of the window, is an important boundary used to differentiate data types. Specific data types refer to Sect. 3.2.

3.2 Definition of Late Data and Straggler

We divide the data into four types, normal pending data, normal data, late data, and stragglers, by sliding time window of low watermark. We can get them by comparing the timestamp of processing data and the range of the sliding time window.

Normal pending data: data with timestamp belonging to the interval [$lowwatermark + STRAGGLERSIZE$, $+\infty$); Normal data: data with timestamp belonging to the interval [$lowwatermark + STRAGGLERSIZE/2$, $lowwatermark + STRAGGLERSIZE$); Stragglers: data with timestamp belonging to the interval [$lowwatermark$, $lowwatermark + STRAGGLERSIZE/2$); Late data: data with timestamp belonging to the interval ($-\infty$, $lowwatermark$).

Every time, late data and normal pending data are outside the time window, stragglers and normal data are in the time window. Every new data appears as normal pending data and the timestamp of new data is after $lowwatermark + STRAGGLERSIZE$. All the types of data are shown as Fig. 2, where A is the late data, B is the stragglers, C is the normal data and D is the normal pending data.

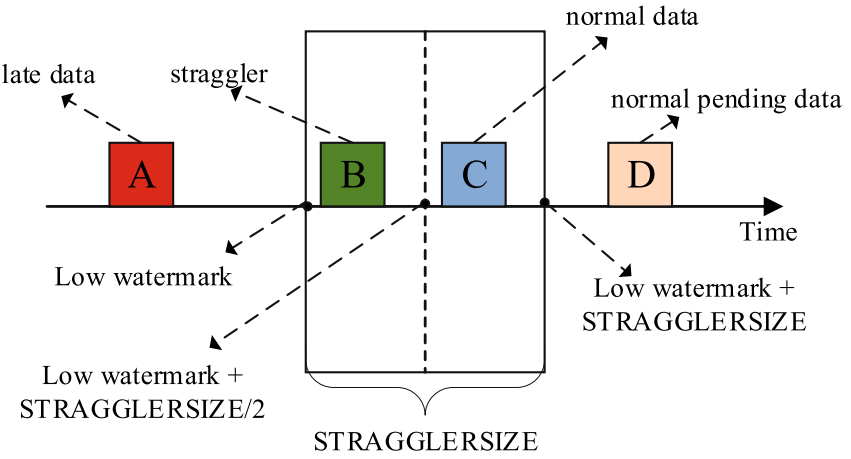

Fig. 2. Data types

3.3 Delayed Stream Arrival Detection

In order to generate a new watermark and keep it updated when new elements processed, we use m, the maximum allowed delay, to represent. And dynamic m (Table 1) is proposed to adapt the real-time stream account for high accuracy and low latency. When the number of stream per second (with *dataRate* for short) bigger than a certain threshold (with $DATARATETHRESHOLD$ for short), dynamic m will change automatically. All the parameters we used to dynamic rectification low watermark are shown in Table 1.

Table 1. Parameters used for watermark generators

Parameter	Description
m	Maximum allowable delay. $m \in [1, SLIDEWINDOWSIZE]$
CHANGERATE	Maximum allowable delay change ratio, CHANGERATE $\in (0, 1]$. Default is 0.01
DATARATETHRESHOLD	Threshold of data per second. Default is 5
dataRate	Number of data per second in real-time stream

And Algorithm 1 describes how m change with $CHANGERATE$ in detail. For a continuous stream, there is a list of collecting new data when it is processed. We get *dataRate* by counting the number of data in the list. Then, we will compare *dataRate* and $DATARATETHRESHOLD$ to decide how to change m. When *dataRate* is higher than $DATARATETHRESHOLD$, m will increase by $CHANGERATE$, otherwise m will reduce. The new m will be used to generate next watermark. The Algorithm 1 shows the detail of the change of m.

Algorithm 1. Adaptive watermark generation

Require:
A data stream S
Ensure:
$m \leftarrow$ 350L; watermark $\leftarrow$ 0;
dataRate $\leftarrow$ 5; $DATARATETHRESHOLD \leftarrow 0.01$;
extractedTimestamp = -∞; lastExtractedTimestamp = -∞;
dataCount = 0; N = 1000; SLIDEWINDOWSIZE = 500L;
List: processedELments $\leftarrow$ NULL;
for each $e \in S$ **do**
  watermark = extractedTimestamp - m;
  emit(watermark)
  processedELments $\leftarrow$ e;
  if processedELments.size () > 0 **then**
    lastExtractedTimestamp = processedElments.get(processedELments.size ()-1)
    for each *element* $\in$ *processedELments* **do**
      if lastExtractedTimestamp >= lastExtractedTimestamp - N && lastExtractedTimestamp <= lastExtractedTimestamp **then**
        dataCount ++
      end if
    end for
    dataRate = *dataCount*/(N/1000)
    dataCount $\leftarrow$ 0
  end if
  if *dataRate* > $DATARATETHRESHOLD$ **then**
    $m = m + m * CHANGERATE$
    if m > SLIDEWINDOWSIZE **then**
      m = SLIDEWINDOWSIZE
    else
      $m = m - m * CHANGERATE$
      if m <= 1 **then**
        m = 1
      end if
    end if
  end if
end for

The initial values of m, $CHANGERATE$ and $DATARATETHRESHOLD$ must be specified when generating watermark with dynamic m. In our experiment, the default values for these parameters are 350, 0.01 and 5. Figure 3 is an example to explain how the watermark works. When the first data which *et* (event time) is 19213 comes, m is 350, *dataRate* is 0, *dataRate* is smaller than $DATARATETHRESHOLD$ and the watermark is $19213 - 350 = 18863$. When second data with *et* 19356 arrives, *dataRate* is 1, which less than 5, and m is reduce to $350 * (1 - 0.01) = 346$, therefore, watermark is $19356 - 346 = 19010$. The third, fourth and fifth data are similar to above. When the sixth data arrive, *dataRate*

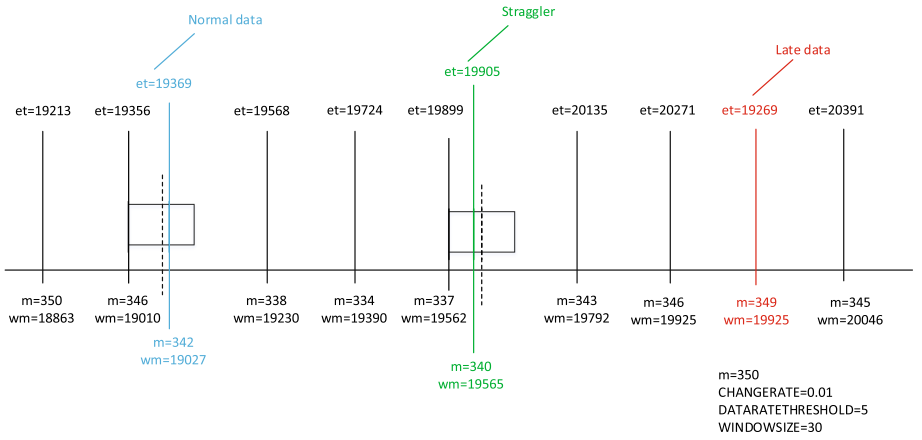

Fig. 3. Example for watermark generation and different data types (Color figure online)

is equal to $DATARATETHRESHOLD$ and m will increase, $334 * (1 + 0.01) = 337$, hence, watermark is $19899 - 337 = 19562$. We can see from our method that m can change dynamically. It can better adapt to the constantly changing in real-time stream. The different data types can be clearly seen from Fig. 3. According to the definition of data types, the blue is normal data. The green is straggler. And the red is late data.

4 Experiments

4.1 Setup and Dataset

Setup: We have implemented low watermark sliding time window with dynamic m on top of Apache Flink v1.9.0. The experiment runs on the computer with 8 GB memory and i5-8500 CPU at 3 GHz.

Dataset: We use three different scale of data sets to compare watermark with dynamic m and static m, the DBES 2012 [10], DBES 2015 [2] Grand Challenge and Out-of-Order dataset from Internet of Things [17]. **1)** The $DBES$ 2012 data set, which has 9564 tuples with 3225 out-of-order data, comes from the high-tech manufacturing equipment. **2)** The $DBES$ 2015 data set is composed of reports of taxi trips including starting location, drop-off location, the time and amount of payment. After deduplicating the value, there are 10427 tuples with 8384 out-of-order data. **3)** The Out-of-Order data set is generated from standard commercial equipment, networks and protocols commonly used in IoT applications. S-8 to S-10 records the WLAN information, whose unordered data is higher than D-1 to D-5, recording the UMTS information. We choose S-10 for our experiment, which has the most data in S-8 to S-10. This data set, including 18386 out-of-order data, has 29999 tuples.

We use the timestamp of every tuple from the three data sets to distinguish different data type, especially stragglers and late data. For the *DBES* 2012 data set, we choose timestamp which is generated by the embedded PC upon the creation of the given event. For the *DBES* 2015, we choose the timestamp of starting point. And for the Out-of-Order data set, we use S-Client-Detection-Time. Figure 4 shows the top 100 tuples of every data set. (a) is *DBES* 2012, (b) is *DBES* 2015 and (c) is S-10. We can see the trend, timestamp increases overall, but there is some disorder, of every data set clearly from the Fig. 4.

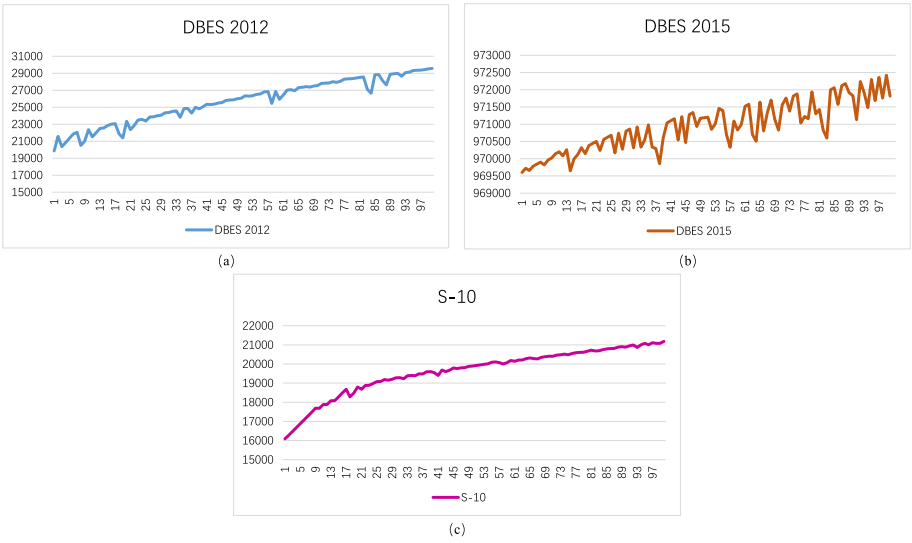

Fig. 4. Trend of data sets

4.2 Comparisons

The ratio of late data (with *Dropped* for short) and the ratio of stragglers are two metrics in our experiment. We compare the two method of static m and sliding time window of low watermark with dynamic m. And we choose two parameters of m and $CHANGERATE$ to change. The results of two different methods are shown in Table 2.

We can see the ratio of dropped and the ratio of straggler from the Table 2. For the static m method, we use 0 and 350 to represent how long allow delay. For the dynamic m, we use an initial value of 350, which can compare with 350 ms delay of static m. From the Table 2, the ratio of *Stragglers* in static m is null, which means the static method cannot find stragglers. On the other hand, our approach not only can find stragglers, but also can decrease the ratio of dropped. That illustrates dynamic m could adapt the change of real-time stream better and the window could find stragglers. For the different $CHANGERATE$, it has

Table 2. Comparison results of different methods

Dataset	Method	m	CHANGERATE	Dropped (%)	Stragglers (%)
DBES 2012	Original	–	–	33.72	–
	Static m	0	–	9.45	–
		350	–	0.31	–
	Sliding time window of low watermark	Dynamic	0.01	0.25	61.23
		Dynamic	0.1	0.25	61.47
DBES 2015	Original	–	–	80.40	–
	Static m	0	–	35.26	–
		350	–	17.16	–
	Sliding time window of low watermark	Dynamic	0.01	12.15	21.31
		Dynamic	0.1	12.11	21.28
S-10	Original	–	–	61.29	–
	Static m	0	–	14.47	–
		350	–	0.14	–
	Sliding time window of low watermark	Dynamic	0.01	0.05	49.15
		Dynamic	0.1	0.05	49.15

almost the same dropped rate and stragglers rate. That is because m is limited by $WINDOWSIZE$ and trend of data sets. The specific analysis of result is as follows.

For the method of static m, different m has different dropped ratio. Every dropped ratio has lower latency than original dropped ratio, but it doesnot distinguish stragglers. And we can see the result of 350 ms delay is better than 0ms, which certifies allowing delay can reduce dropped ratio. As we can see from Table 2, for the $DBES$ 2012, original dropped ratio is 33.72%. When $m = 0$, dropped ratio is 14.47% and when $m = 350$ is 0.31%. $DBES$ 2015 and S-10 data sets have similar result. The larger m the lower dropped ratio. As a result of allowed delay, it can process more data.

For the dynamic m, we can see from the Table 2 that dropped ratio of dynamic m is lower than static m. Moreover, our method can distinguish stragglers. $CHANGERATE$ determines the magnitude of each change, but it isnot the only factor. Figure 4 describes that the trend of data set also has influence on the result. In Fig. 4 (a), large changes occur in an intervals in $DBES$ 2012. Uncertain changes will have some impact on the results, but it has small influence. From the Fig. 4 (a), we can see the trend of $DBES$ 2012 data set is that there is a big change in every other period of time, which has small impact. The larger the value of $CHANGERATE$, the higher the tolerance of the system.

Hence, when $CHANGERATE$ is 0.1, straggler ratio is higher than it is 0.01. From the Fig. 4 (b), we can see the trend of $DBES$ 2012 data set is that there are more fluctuations. Too much out-of-order data and irregular changes will have some impact on the results. A little change of watermark will influence the result. That results in the greater $CHANGERATE$ the lower straggler rate. In the Fig. 4 (c), the timestamp grows steadily. Stable increasing timestamp has almost no influence in different $CHANGERATE$. Moreover, occasional changes between nearby data provide a guarantee for stream processing. The above reasons make the results the same in different $CHANGERATE$.

5 Conclusion

In this paper, we propose a new method called sliding time window of low watermark to generate a watermark dynamically and distinguish stragglers and late data. The number of data in unit time, the threshold and the ratio of change is three parameters to decide what the value of new watermark will be. Then we use the new watermark and the sliding time window to distinguish normal data, late data and stragglers. Our method adapts the perplexing change of stream better than static m watermark generation, as the experiment result show. Moreover, detailed data classification will be beneficial to stream processing.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (61772231), the Shandong Provincial Natural Science Foundation (ZR2017MF025), the Project of Shandong Provincial Social Science Program (18CHLJ39), the Science and Technology Program of University of Jinan (XKY1734 & XKY1828), and the Project of Independent Cultivated Innovation Team of Jinan City (2018GXRC002).

References

1. Akidau, T., et al.: MillWheel: fault-tolerant stream processing at internet scale. Proc. VLDB Endow. **6**(11), 1033–1044 (2013)
2. Awad, A., Traub, J., Sakr, S.: Adaptive watermarks: a concept drift-based approach for predicting event-time progress in data streams. In: EDBT, pp. 622–625 (2019)
3. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in Apache Flink®: consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (2017)
4. Carbone, P., et al.: Large-scale data stream processing systems. In: Zomaya, A.Y., Sakr, S. (eds.) Handbook of Big Data Technologies, pp. 219–260. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-49340-4_7
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. **36**(4), 28 (2015)
6. Chen, Q., Liu, C., Xiao, Z.: Improving MapReduce performance using smart speculative execution strategy. IEEE Trans. Comput. **63**(4), 954–967 (2013)

7. Grulich, P.M., Saitenmacher, R., Traub, J., Breß, S., Rabl, T., Markl, V.: Scalable detection of concept drifts on data streams with parallel adaptive windowing. In: EDBT, pp. 477–480 (2018)
8. Hwang, J.H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., Zdonik, S.: High-availability algorithms for distributed stream processing. In: 21st International Conference on Data Engineering (ICDE 2005), pp. 779–790. IEEE (2005)
9. Iqbal, M.H., Soomro, T.R.: Big data analysis: Apache storm perspective. Int. J. Comput. Trends Technol. **19**(1), 9–14 (2015)
10. Jerzak, Z., Heinze, T., Fehr, M., Gröber, D., Hartung, R., Stojanovic, N.: The debs 2012 grand challenge. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 393–398. ACM (2012)
11. Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., Maier, D.: Out-of-order processing: a new architecture for high-performance stream systems. Proc. VLDB Endow. **1**(1), 274–288 (2008)
12. Nagano, K., Itokawa, T., Kitasuka, T., Aritsugi, M.: Exploitation of backup nodes for reducing recovery cost in high availability stream processing systems. In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium, pp. 61–63. ACM (2010)
13. Shah, M.A., Hellerstein, J.M., Brewer, E.: Highly available, fault-tolerant, parallel dataflows. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 827–838. ACM (2004)
14. Shoro, A.G., Soomro, T.R.: Big data analysis: Apache spark perspective. Glob. J. Comput. Sci. Technol. **15**, 7 (2015)
15. Toshniwal, A., et al.: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 147–156. ACM (2014)
16. Traub, J., et al.: Scotty: efficient window aggregation for out-of-order stream processing. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1300–1303. IEEE (2018)
17. Weiss, W., Jiménez, V.J.E., Zeiner, H.: A dataset and a comparison of out-of-order event compensation algorithms. In: IoTBDS, pp. 36–46 (2017)
18. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 423–438. ACM (2013)