ATLANTIS PRESS

Advances in Intelligent Systems Research, volume 134

2nd International Conference on Control, Automation, and Artificial Intelligence (CAAI 2017)

A Highly-Efficient Fault Tolerance Method for An Scalable Stream Processing System

Guanghui Chang[1], Peizhen Li[2,*] and Guangxia Xu[1,3]

[1]School of Software Engineering, Chongqing University of Posts and Telecommunications, Chongqing, China

[2]School of Information and Communication Engineering, Chongqing University of Posts and Telecommunications, Chongqing, China

[3]Information and Communication Engineering Postdoctoral Research Station, Chongqing University, Chongqing, China

*Corresponding author

***Abstract*—An effective fault-tolerant mechanism becomes essential to the credible stream processing system (SPS). However, the traditional introduction of fault-tolerant mechanisms has a relatively negative influence on the systemic calculation efficiency. And enhancing the system scale out is an effective way to solve this problem. This paper proposes a highly-efficient fault recovery method that is for an extensible SPS. On the one hand, based on a reasonable partition and upstream backup of its internal calculation status. The SPS can monitor the bottleneck state of the operators and then migrate the upstream backup state of the operators that is in need of scale out to the new nodes to achieve dynamic scale out of the SPS; On the other hand, when a node fault in the SPS, the system can dynamically expand a secondary node for the erroneous one, and then use the upstream backup algorithm to achieve fault recovery. In this paper, a mathematical model is built for the SPS. And based on this, the fault recovery time and computational efficiency of the rapid fault tolerant method are analyzed. Experimental results have shown that the one proposed in this paper is more effective in fault tolerance which gives the system a higher calculation efficiency by reducing the frequency of upstream backup of node state.**

Keywords—SPS; dynamic scale out; fault tolerance

I. INTRODUCTION

In recent years, big data stream computing has gained increasing popularity in all walks of life [3], such as the smart grid [4], finance and banking, Internet and internet of things, etc. In such an era of data explosion, business can efficiently extract timely information from massive stream data through stream processing system. While in the academic circles, the SPS has also garnered high attention in various top international academic conferences.

Although the SPSs have improved the timely process of big data and gained wide application in the industrial circles as well, the problem of highly-efficient fault recovery has always been a major obstacle in the way of the SPSs. In the SPSs, especially the stateful operator, it requires longer time for fault recovery in nodes and it is possible to lose calculation state in the nodes, which is intolerable for the efficient and credible SPSs. Therefore, the academic community has put forward a number of fault-tolerant programs based on the SPSs [8-14, 19, 20], most of which are achieved by backing up operator state or the calculation tuples. For instance, Zhang Z et al. [19] has reduced both the overhead and latency of the system by mixing the active standby and passive standby, utilizing the ideas of the optimal fault tolerance. Upadhyaya P et al. [20] has implemented different fault-tolerant algorithms for different operators so that the recovery time and fault tolerance efficiency can achieve optimal. Fernandez R C et al. [7] has achieved fault tolerance on the basis of dynamic scale out. This kind of fault-tolerant thinking is a novel integration of dynamic scale out and fault tolerance. Zoe. S et al. [9] has realized the systemic fault tolerance by the upstream backup of the calculation tuples and periodically checkpoint of the operator state. But different from conventional methods, it partitions the state of the operators into multiple windows. In the process of state checkpoint, the operators will not be frozen; instead, the state of each window will go through the process of asynchronous backup, thereby improving the computational efficiency of the system.

Moreover, due to the suddenness and load fluctuation [3] of the system in the massive streaming data as well as relatively substantial dynamic changes of the system load, the capability of dynamic scale out of the SPS [2,6,7,15] appears extremely important. The current SPSs such as Apache spark streaming [5], Twitter storm [16] and S4 [17] are all realized by static configuration, but there are deficiencies in terms of dynamic scale out.

However, to achieve dynamic scale out and fault recovery in the SPSs, it takes up relatively large system resources. Thus, Fernandez R C et al. [6, 7] has proposed a method integrating the dynamic scale out and fault recovery. Based on the partition of query state of operators and the periodical upstream backup, the upstream backup operators can be just migrated to a new node when it is in need of scale out; when an node fault has occurred in the system, the system just need to dynamically expand a secondary node for the erroneous one and reproduce a new tuple that is not calculated. However, this kind of method requires a periodical upstream backup for the query state of the operators. Since the relevant operators must be frozen in the process of the upstream backup, this kind of backup strategy can greatly affect the computational efficiency of the system. Therefore, a more efficient synthetic method is proposed in this paper. The query state of the operators can be divided into three parts. Based on the upstream backup of these three parts, when a certain node is in need of dynamic scale out, the node just

smoothly move its upstream backup state to the new node. The backup state of the nodes can be updated accordingly in the wake of the changes of the topological structure in the system. The policy for updating state in this paper is that only the node state changes, that are after its dynamic scale out, will the system update the upstream backup state of that node. When fault occurs in a certain node, the system will dynamically expand a secondary node with high availability algorithm for upstream backup, and then perform the upstream backup algorithm on the backup node so as to meet the demand of the system for fault tolerance.

The contributions of this paper can be illustrated as follows:

1. By improving the state of the operators upstream backup strategy in the system, the computational efficiency of the system can be effectively improved based on the dynamic scale out of the SPSs.

2. By integrating rollback recover method with dynamic scale out, the rollback recover method can be optimized and secondary node of the original method can be omitted, so as to achieve the fault tolerance and save hardware resources as well.

In the following part, specific problems are analyzed in this paper; section three focuses on the management strategy of the query state of the operators; section four and five are the highlights of this paper, which introduces the dynamic scale out and fault tolerance approach respectively; section six makes confirmatory contrasts between the methods and reference discussed in this paper; and section seven is about the related work.

II. System Model

Definition 1: data model. In this paper, the data stream (s) is regarded as a series of consecutive and ordered tuples (t), and the relationship between them can be expressed as $t \in s$. And one of the tuples t= (τ, key, values). In the triples, τ represents the time stamp, key is the key of the tuples and values the effective load.

Definition 2: query model. The directed acyclic graph can be expressed as q= {O, S}. A circle indicates an operator and the lines between operators represent the data stream.

Definition 3: query execution. The dynamic scale out has occurred during the process of the system, among which the operator O_1 has been scaled out into O_1^1 and O_1^2. The dynamic scale out of the operator O_i can be expressed as $O_i \rightarrow \{O_i^1, O_i^2, \ldots, O_i^n\}$, and $n \in N^+$.

As shown in Figure 1, the SPS can be divided into five modules in this paper: the calculation module, the fault recovery module, the dynamic scale out module, the system monitoring module and the pre-processing resource pool. Among them, the calculation module of the SPS is mainly responsible for the query task of the system. By monitoring the calculation state of each operator within the system, the system monitoring module can determine whether there is a need for dynamic scale out to improve computational efficiency, or whether a certain node fault has occurred in the system. The fault recovery module and the dynamic scale out module are combined with each other to realize fault recovery of the system. When it is in need of dynamic scale out for the relevant operators, the system will access the corresponding node resources from the computing resource pool.

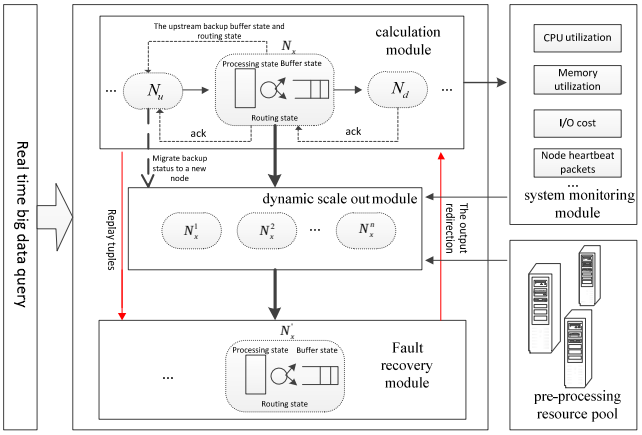

FIGURE I. SPS ARCHITECTURE

III. The Management for Query State of Operators

The partition of the query states of the operators, cited from the literature [7], is introduced in this paper, which are processing state, routing state and buffer state respectively. Corresponding operations can be performed on the query state respectively so as to realize features of scale out and fault-tolerance. Among them, processing state, expressed as κ_0, represents the tuple state that has already been calculated or is waiting to be polymerized or merged. Routing state refers to the routing status of operators, which can be expressed as ρ_o. Buffer state is the state of output queue of operators, which can be expressed as β_o. It is emphasized that this paper deals with the stateful stream, so the existing processing state makes the description of the system more clearly. But due to the changes in the upstream backup strategy, only routing state and buffer state will be backed up in the backup algorithms in this paper.

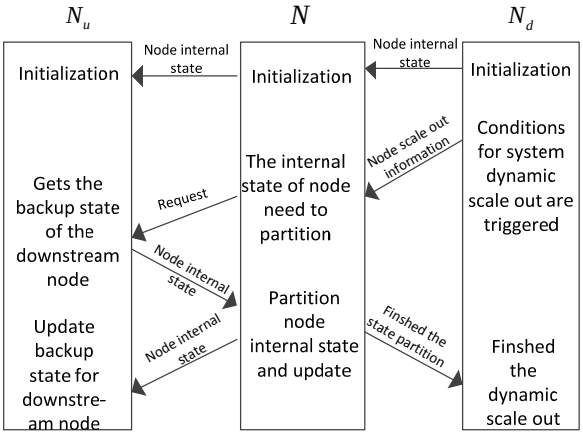

FIGURE II. BACKUP AND PARTITION OF NODE STATE

On the basis of node state partitioning, it is necessary to do some fine-grained operations on the node internal state to meet the demands of system dynamic scale out and fault recovery. These operations mainly include checkpoint state, backup state and partition state.

Checkpoint state. In order to achieve the upstream backup of the internal state of a node, it is necessary to checkpoint. As shown in Figure 2, the node in the initialization process will be checked, and the state of the checkpoint will be stored in the upstream node. The upstream backup state includes routing state ρ_o and β_o buffer state in the downstream.

Backup state. In order to achieve the dynamic scale out, it is necessary to make regular upstream backup of the internal state of the nodes. Figure 2 shows the upstream backup strategy: when the system initialization is completed, the internal state of each node will be backup in its upstream node. After the node N_d is scale out, the N, ρ_o and β_o will vary with the corresponding scale out. In order to maintain the consistency of the nodes, the upstream backup state of the nodes is required. In particular, when the upstream nodes are more than one, how do we choose the upstream backup node? We randomly select a node as a backup node $hash(id(N)) \bmod |up(N)|$ among its many upstream nodes, where we denote $|up(N)|$ as the total numbers of its upstream nodes, $id(N)$ as its unique identifier such as MAC address, etc.

Partition state. In order to achieve the dynamic scale out, the internal state of nodes must partition. As shown in Figure 2, firstly, when the N_d completes the dynamic scale out, it will send scale out information to the N. The scale out information mainly includes the number of extended nodes and so on. Secondly, when the node N receives the scale out information, it will send a request to the upstream backup node N_2 to obtain the state of the backup in the upstream node, partition the corresponding state according to the scale out information and set a new state on the node N. Thirdly, send a confirmation message to the downstream node N_d indicating that the state segmentation of the node has been completed.

IV. SCALE OUT AND FAULT TOLERANCE

Based on the research above, this paper will then elaborate on the system's dynamic scale out. First of all, it introduces when the operator can be scaled out in the system. In this paper, he monitoring module is included in SPS, each operator has a number of nodes, we randomly select a bottleneck node in a operator. The monitoring module will periodically reflect the CPU utilization in the bottleneck node. The provision is that if the CPU utilization of the bottleneck node in two consecutive cycles outweighs the threshold we set, dynamic scale out will be applied to the operator where it happens.

When the monitoring module in two consecutive cycles detects operator at a bottleneck node where CPU utilization exceeds the threshold set by the system, the monitor module will identify the number (n) of operators for scale out according to the CPU utilization rate, and send extended message to the operator O. Then the operator O sends a request to upstream operator O_u^i which backup its state, after which the system conducted the $get\_operator\_state(O)$ operation and gets its query status. Afterwards, the state of the new node is set to the state of the operator O and the buffer of the newly added nodes is emptied. Because what we calculate in the system is stateful stream and the buffer storage can be sent to the downstream node after the completion of the operator calculation, the contents of the bottleneck node which already exists may be sent to the downstream node. If the contents were not emptied, the results through the system calculation may be interfered. Finally, the operator O sends a ack message to the monitoring module as a confirmation of the extended completion.

Backup upstream algorithm is a kind of high availability algorithm, which can effectively achieve the fault recovery in the SPS, for specific information please refer to the literature [14]. However, backup upstream algorithm can be a serious waste of hardware resources because each node needs a secondary node in SPS. By integrating the dynamic scale out and upstream backup algorithm, this paper proposes a method that can effectively avoid the disadvantages of backup algorithm upstream. When a node needs fault recovery, it only needs a dynamic scale out to a secondary node. Figure 6 shows the specific fault tolerance process. In this paper, not all nodes are distributed with a secondary node, but when fault occurs in any arbitrary node, the system will expand a fault-tolerant node from the pre-processing resource pool, and then call the upstream backup algorithm for system fault tolerance.

V. EVALUATION

In this section, a comparison is made between the method proposed in this paper and the method in literature [7] in terms of computational efficiency and fault recovery time.

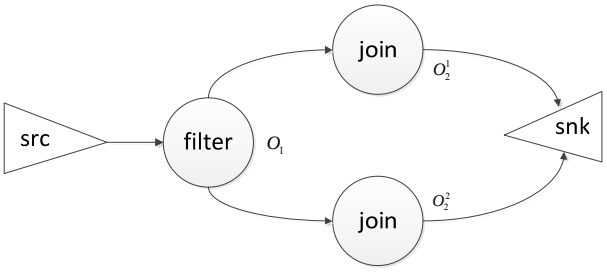

FIGURE III. EXPERIMENTAL SETUP

This section will briefly describe the experimental results mentioned in this paper, and the process of the experiment can be divided into two parts. Firstly (6.2-a), through manually injecting anomalies in SPS and changing the input tuples rate, we can collect fault recovery time in SPS, and then compare the differences of two methods in fault recovery time. Secondly (6.2-b), through manually controlling dynamic scale out and the fault frequencies in the SPS, we can compare computational efficiency under two method. As can be seen from Figure 3, the experimental cluster consists of two operators. Specifically, operator 1 carries on the preliminary filter to the received tuple, the operator 2 to further join and store the result to snk Moreover, operator 1 is composed of a node and operator 2 is composed of two nodes, and the operating environment is Ubuntu14.04. And meanwhile, each node has a 4 core CPU and

an 8G memory. Through manually injecting anomalies and manually setting of dynamic extended threshold, the system can satisfy condition for its dynamic scale out and fault recovery.

A Fault Recovery Time

In the query process, input tuple rate is increased from 100tuples/s to 1000tuples/s. Errors are manually injected to one node in operator 2 during the process, and the recovery time of SPS is observe under different input rate. As is shown in Figure 4, with the increase of tuple input rate, the fault recovery time of LSOM is getting longer, while the fault recovery time of HSOM is stable. When the input rate of the tuple reaches 900tuples/s, the fault recovery time of LSOM is longer than the fault recovery time of the HSOM.

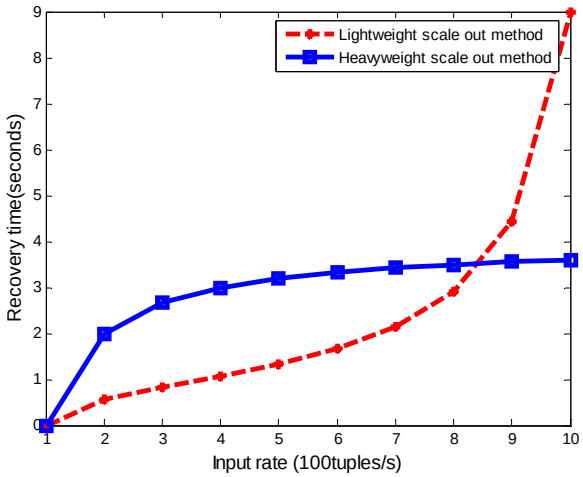

FIGURE IV. COMPARISON OF FAULT RECOVERY TIME

B The Computational Efficiency Of The SPS

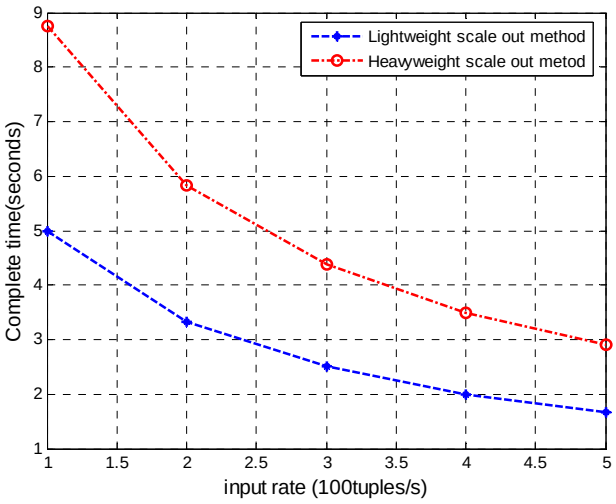

FIGURE V. THE COMPARISON OF TWO METHODS FOR PROCESSING THE EFFICIENCY OF THE CASE WITHOUT

During the whole process of the query, if the whole SPS is made to run smoothly without any dynamic scale out or fault, experimental results can be got in Figure 5. The vertical axis represents the completion time of the query, while the horizontal axis indicates the input rate of tuples. As can be seen from Figure 5, the calcultation efficiency of LSOM is higher than that of HSOM.

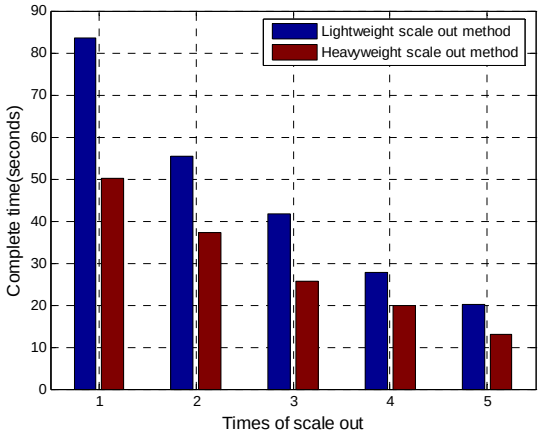

FIGURE VI. THE COMPARISON OF TWO METHODS FOR PROCESSING EFFICIENCY OF THE CASE WITH FAULT TOLERANCE AND SCALE OUT

By manually adjusting the CPU threshold value of one node in operator 2, the system can add only one node when realizing dynamic expansion during the calculation. In Figure 6, the horizontal axis represents the frequency during the dynamic scale out of the query process, while the vertical axis stands for the query time for completion. As can be seen from the figure above, with the increase of dynamic scale out, the calculation efficiency of both methods has improved greatly, and the computational efficiency of LSOM is slightly higher than that of the HSOM.

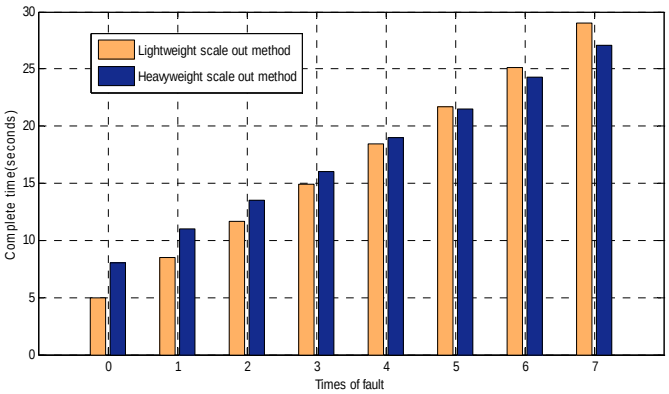

FIGURE VII. THE COMPARISON OF TWO METHODS FOR PROCESSING THE EFFICIENCY OF THE CASE HAVE SCALE OUT

During the query process, anomalies are manually injected in operator 2. In Figure 7, the horizontal axis indicates fault frequency of the operator 2 in a process of calculation, while the vertical axis represents the time for query completion. From Figure 7, it can be seen if the frequency of anomalies is 5 times less than that in arbitrary nodes of operator 2, the computational efficiency of LSOM on average can exceed that of scale out method at 22.8%; however, when the frequency of

anomalies is 5 times more than that in arbitrary nodes of operator 2, the computational efficiency of LSOM will be at a disadvantage.

VI. CONCLUSIONS

Based on the dynamic scale out, the paper aims to make effective use of high availability fault tolerant algorithm to realize fault tolerance of the system. The use of lightweight backup strategies in the process of state backup can improve the computational efficiency of the system. Moreover, concerning the method proposed in this paper, the experimental results show that in the low error rate SPS the computational efficiency is relatively higher. However, for anomalies that occurs in both the upstream and downstream nodes, the fault tolerance method mentioned in this paper can hardly realize any effective fault tolerance. In the future, we intend to further improve the method mentioned in this paper, making it possible to deal with the anomalies occurred simultaneously in both the upstream and downstream nodes.

ACKNOWLEDGMENT

The authors acknowledge support from the National Natural Science Foundation (grant no.61309032 and 61272400); the Program for Innovation Team Building at Institutions of Higher Education in Chongqing (grant no. CXTDG201602010); the China Postdoctoral Fund (grant no. 2014M562282); the Project Postdoctoral Supported in Chongqing (grant no. Xm2014039); the Wenfeng Leading Top Talent Project in CQUPT, the New Research Area Development Programme (grant no. A2015-44); the Science and Technology Research Project of Chongqing Municipal Education Committee (grant no. KJ1400422, KJ1500441 and KJ1400431); the Common Key Technology Innovation of Important Industry by Chongqing Science and Technology Commission (grant no. CSTC2015ZDCY-ZTZX40001); the Collaborative Innovation Center for Information Communication Technology (grant no. 002); the Social Livelihood Science and Technology Innovation Special Projects of Chongqing (grant no. cstc2016shmszx40001); the University Outstanding Achievements Transformation Funding Project of Chongqing.

REFERENCES

[1] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013, pp. 423-438.

[2] K. Karama, R. Matteo, A. David, M. V. D. Schaar, "Low Power and Scalable Many-Core Architecture for Big-Data Stream Computing," VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on IEEE, 2014, pp. 468-473.

[3] D. W. Sun, G. Y. Zhang, W. M. Zheng, "Big Data Stream Computing: Technologies and Instances," Journal of Software, 2014.

[4] R. C. Fernandez, M. Weidlich, P. Pietzuch P, A Gal, "Scalable stateful stream processing for smart grids," Matthiasweidlich Net, 2014.

[5] M. Zaharia, T. Das, H. Li, S Shenker, I Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," Hotcloud, 2012, pp. 10-10.

[6] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, "Scalable and Fault-tolerant Stateful Stream Processing," OpenAccess Series in Informatics (OASIcs), 2013, pp. 11-18.

[7] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," ACM SIGMOD International Conference on Management of Data. ACM, 2013, pp. 725-736.

[8] H. Chai, W. Zhao, "Fault Tolerant Event Stream Processing for Autonomic Computing," Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference onIEEE, 2014, pp. 109 - 114.

[9] Z. Sebepou, K. Magoutis "CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators," Dns Twitter, 2011, 6548(5), pp. 145-156.

[10] K. B. Muralidharan, G. S. Kumar, M. Bhasi, "Fault tolerant state management for high-volume low-latency data stream workloads," Data Science & Engineering (ICDSE), 2014 International Conference on. IEEE, 2014, pp. 24-27.

[11] A. Martin, T. Smaneoto, T. Dietze, A. Brito, C. Fetzer, "User-Constraint and Self-Adaptive Fault Tolerance for Event Stream Processing Systems," Dependable Systems and Networks (DSN), 2015, pp. 462 – 473.

[12] M. Balazinska, J. H. Hwang, M. A. Shah, "Fault-Tolerance and High Availability in Data Stream Management Systems," Springer US, 2009.

[13] J. H. Hwang, Y. Xing, U. Cetintemel, S. Zdonik, "A Cooperative, Self-Configuring High-Availability Solution for Stream Processing," Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on. IEEE, 2007, pp. 176-185.

[14] J. H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M Stonebraker, "High-Availability Algorithms for Distributed Stream Processing," Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference onIEEE, 2005, pp. 779-790.

[15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," IEEE Transactions on Parallel & Distributed Systems, 2012, 23(12), pp. 2351-2365.

[16] N. Marz. Twitter storm. https://github.com/nathanmarz/storm/wiki, 2017.

[17] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, "S4: Distributed Stream Computing Platform," Data Mining Workshops (ICDMW), 2010 IEEE International Conference onIEEE, 2010, pp. 170-177.

[18] Apache. Apache hadoop. http://hadoop.apache.org, 2017.

[19] Z. Zhang, Y. Gu, F. Ye, H. Yang, M Kim, "A Hybrid Approach to High Availability in Stream Processing Systems," Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference onIEEE, 2010, pp. 138-148.

[20] P. Upadhyaya, Y. C. Kwon, M. Balazinska, "A latency and fault-tolerance optimizer for online parallel query plans," Proceedings of the 2011 ACM SIGMOD International Conference on Management of dataACM, 2011, pp. 241-252..