Fast and Precise recovery in Stream processing based on Distributed Cache

Yingying Zheng
Institute of Software, Chinese Academy of Sciences
zhengyingying14@otcaix.iscas.ac.cn

Wei Wang*
Institute of Software, Chinese Academy of Sciences
wangwei@otcaix.iscas.ac.cn

Lijie Xu
Institute of Software, Chinese Academy of Sciences
xulijie@otcaix.iscas.ac.cn

Zhen Tang
Institute of Software, Chinese Academy of Sciences
tangzhen12@otcaix.iscas.ac.cn

Zhongshan Ren
Institute of Software, Chinese Academy of Sciences
renzhongshan13@otcaix.iscas.ac.cn

Jun Wei
Institute of Software, Chinese Academy of Sciences
wj@otcaix.iscas.ac.cn

Dan Ye
Institute of Software, Chinese Academy of Sciences
yedan@otcaix.iscas.ac.cn

ABSTRACT

Stream processing system (SPS) faces the problem of node failure when running over a long period of time. In addition, "exactly once" precise semantic guarantee is more and more important for SPS in some scenarios. In general, the approaches to achieve precise semantic is by using global snapshot, which should store state and records to external reliable storage or rely on transactions. However, these approaches suffer from high recovery latency, because of large I/O disk overhead. In order to reduce excessive latency in failure recovery, we save the intermediate results which are produced during the stream processing, and propose an algorithm DCAS which asynchronously snapshots state to implements precise recovery. In addition, we use in-memory distributed cache to provide the storage of intermediate results and snapshots to reduce recovery latency. We evaluate our failure recovery approach in recovery latency and runtime overhead. The experimental results show that our approach is 2 to 6 times faster than other conventional failure recovery approaches, and induces a 6% runtime overhead.

CCS CONCEPTS

• Information systems → Complex Event Processing and Data Streams • Computing methodologies → Distributed and Grid Data Management.

* corresponding author

Internetware'17, September 23, 2017, Shanghai, China

ACM ISBN 978-1-4503-5313-7/17/09…$15.00
https://doi.org/10.1145/3131704.3131724

KEYWORDS

stream processing; failure recovery; distributed cache; precise semantic guarantee; storage management

ACM Reference format:

Yingying Zheng, Wei Wang, Lijie Xu, Zhen Tang, Zhongshan Ren, Jun Wei, Dan Ye. 2017. Fast and Precise recovery in Stream processing based on Distributed Cache. In *Proceedings of Internetware'17, Shanghai, China, September23, 2017,* 6 pages.

https://doi.org/10.1145/3131704.3131724

1 INTRODUCTION

Due to the big data [1] becomes one type of the most popular technology in our era, big data processing applications are springing up in modern high-tech world. These application scenarios are now widely used in stream processing system (SPS) [4-5]. However, as the scale of SPS cluster increases, the probability of node failure becomes high.

According to the semantic guarantee, failure recovery in SPS can be split to the following three categories [3]: 1) "at-most-once", "at-least-once" and "exactly once". Nowadays, there are more urgent needs for "exactly once" semantic in the key business fields, such as finance and transportation. In order to ensure the higher precision of semantic guarantee, SPS needs to do more operations for state and records**,** which is difficult to make tradeoffs among runtime overhead, recovery latency and system resource costs in the current SPS. In view of the importance of system performance, we focus on lower runtime overhead and lower recovery latency. However, the existing failure recovery approaches which guarantee "exactly once" semantic can provide low runtime overhead but ignore the recovery latency, such as Apache Flink [9,10, 21]. The reasons of high recovery latency are as follows. First, it generates excessive overhead of disk I/O because of retrieving the latest snapshot from the external reliable storage. Second, it produces high recovery latency, especially

with a mass of operators, because it should take out records from data source.

In this work, we focus on reducing the recovery latency for "exactly once" semantic guarantee. We save intermediate results to support upstream backup, and it can provide operator level recovery. Then we propose DCAS, an asynchronous snapshot algorithm based on distributed cache, to provide information which recovery needed during the stream processing. Furthermore, we choose distributed cache to store this useful information, namely intermediate results and asynchronous snapshot. After failure node is restarted, DCAS regains state of each operator in cluster, and uses the offset to determine the start position of replaying in distributed cache. Furthermore, if the node which receives input stream from data source failed, it can retrieve input stream from distributed cache instead of external persistent storage. More importantly, only the failed node should be restarted and other nodes can operate normally.

We have applied distributed cache to stream processing engine and establish a detailed set of experiments to illustrate the effectiveness of our proposed approach. Experimental results show that, our approach achieves 2-6x faster recovery than other conventional failure recovery approaches which can also guarantee "exactly once" semantic. In addition, the impact of runtime overhead is no more than 6%. In summary, this paper makes the following contributions:

1) We offer storage of intermediate results to provide the upstream backup for failure recovery in operator level.

2) We propose an asynchronous snapshot algorithm based on distributed cache which guarantees "exactly once" semantic and accelerates the recovery speed of state and records significantly.

3) We provide a data storage strategy based on in-memory distributed cache to support the storage of intermediate results and asynchronous snapshot to implement the fast recovery.

The rest of this paper is organized as follows: In Section 2, we introduce the background. Section 3 describes data storage strategy. Section 4 describes the details of our DCAS algorithm followed by Section 5 which gives our recovery scheme. In Section 6, we depict our evaluation. Then, Section 7 gives related work and we finish with conclusion in Section 8.

2 BACKGROUND

2.1 Stream Processing Model

Stream Processing Model (SPM) [2] is used to handle continuous data stream in a certain period of time, and it includes processing units and data streams to be processed. Wherein the data stream is a sequence of tuples which are unbounded in time, and it can be expressed as (a1, a2, ..., an, t) where the ai denotes a record and the t denotes time; A processing unit is a logic operator for data stream, and their processing steps are as follows: each operator 1) obtains input data from the respective input queue, 2) uses its state to do calculations with the input data and return the results, 3) transmits the calculation results to the output queue.

SPM execution model is based on the existing SPS, such as Apache Spark Streaming [12,13,20], Apache Flink [9]. The execution model uses delayed processing to deal with stream, and data window, which can be denoted as $\{d_1, d_2, \ldots d_n\}$ driven by data, to execute aggregation operations. Each host (a physical machine or a virtual machine) can perform multiple concurrent tasks through multi-thread, and each task instance can contain one or more operators, meanwhile there are correlations between upstream and downstream task instance.

2.2 Failure recovery Model

In failure recovery model, we make the following assumptions: 1) we only consider the failures such as software bugs, hardware errors, node failures which make task nodes to not work; 2) after the failure of task node, it cannot consume its upstream records, or output results to downstream task nodes, and its state also cannot be accessed.

The existing failure recovery method has ABS algorithm which is proposed by Apache Flink [10]. It adds barriers to data source and blocks input channels until all barriers are received, and then triggers snapshot to record state. When a node is failure, ABS algorithm needs to retrieve the latest snapshot from the external reliable storage, and replays records from the data source. In SPS, there are higher priority requirements on low runtime overhead and low recovery latency [2]. However, ABS algorithm does not make sure low recovery latency. We improve it by distributed cache to achieve low recovery latency with low runtime overhead.

3 DATA STORAGE STRATEGY

We provide a data storage strategy that using in-memory distributed cache to provide an efficient and reliable storage for fast failure recovery. In-memory distributed cache allows the cache to span multiple servers so that it can grow in size and in transactional capacity [22]. The biggest advantage of in-memory distributed cache is the ability to quickly read and write data. In the cluster of distributed cache, each node has data partition and stores only part of the data. In addition, it also provides a number of data backup of other nodes. Considering the advantages of distributed cache, we store intermediate results and data source to specific distributed cache for different data structure.

1） Cache intermediate results

Due to the need of records sequence, we use distributed in-memory list to guarantee data sequence and reliability of records. In order to cache them, we provide listener mechanism to listen the upstream records. First, operators will store records to the distributed in-memory list, when they obtain records from the upstream operator. Then, operators process these records with specific function. Finally, the new intermediate results will be transmitted to the downstream operator. With the help of backup mechanism, distributed in-memory list can offer the repeated consumption in the operator layer instead of from the data source, which can reduce the failure recovery time.

2） Cache data source

Most SPSs use Apache Kafka [14], a distributed message queues, which provides high throughput to provide reliable guarantee for data source. Considering in-memory distributed

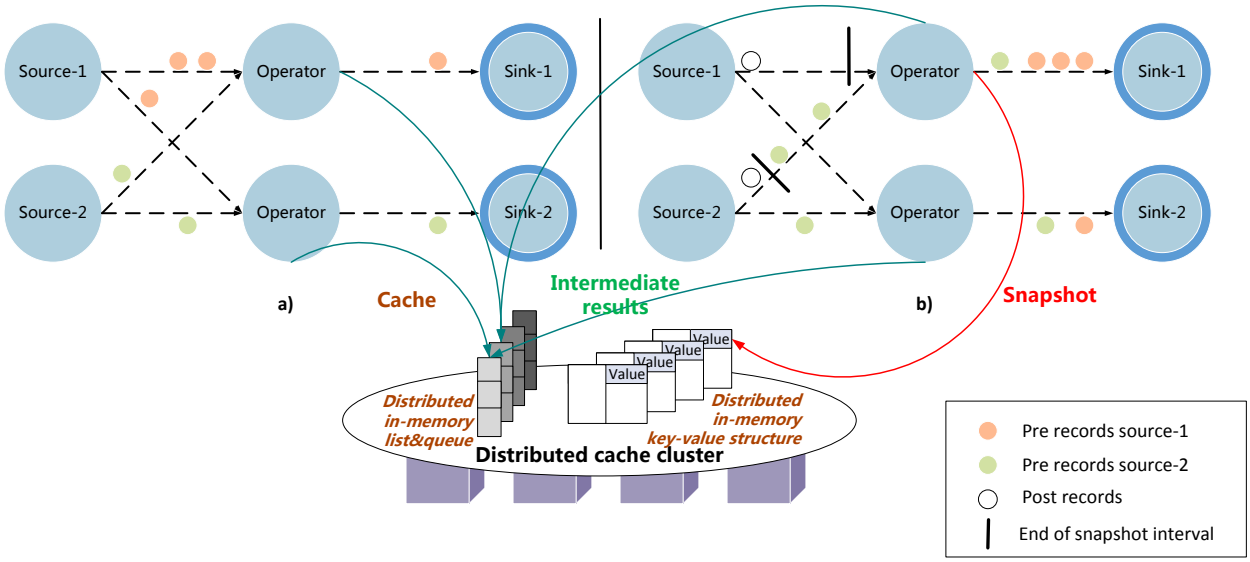

Figure 1. Distributed cache asynchronous snapshot

cache has the ability of quickly fetching data, we use a mixture of Kafka and in-memory distributed cache storage policy.

We still use Kafka message queue as a reliable storage for data source, and distributed cache catch records from Kafka. When recovering from a failure, we can use the backup in distributed in-memory queue to regain original records quickly.

4 DISTRIBUTED CACHE ASYNCHRONOUS SNAPSHOT

4.1 Problem Definition

We define our asynchronous snapshot as S = {*offset**, *states**}. The *offset** denotes offsets which are used to mark the position in distributed cache. The *states** represents all information for failure recovery, including the operator state (*OS*) and user-defined state (*US*). At the same time, we define the concept of *micro-window*, which determines the size of a snapshot interval and the overhead costs in failure recovery during runtime. In addition, it also determines the number of replaying events and recovery latency.

4.2 DCAS Algorithm

Based on the above definition, DCAS algorithm provides operator level failure recovery by means of asynchronous snapshot. Distributed cache helps DCAS to guarantee "exactly once" semantic and achieves fast failure recovery. The core idea of our algorithm is when we maintain continuous data processing, we trigger checkpoint in a snapshot interval; we store state in distributed in-memory key-value structure, and provide "exactly once" semantic by the offset in distributed in-memory list.

In our algorithm (depicted in Algorithm 1), firstly, we should initial input stream *IQ*, output stream *OQ* and the function *F* which is used to process records. In addition, we register listeners which listen to the end of a snapshot interval and trigger checkpoints. A snapshot of different operators is taken independently, and the snapshot information is continuously updated which wouldn't generate too much space overhead. The snapshot contains operator state *S_o*, user-defined state *S_u* of an operator *OO* and the offset *OF* of records *R*. At the same time, they are taken by the function *snapshot*(*S_o*,*S_u*,*OF*) to distributed in-memory Map *M*. In addition, there is a function *listener*(*OO*) which is used to trigger snapshot and cache the processed records *R* by distributed in-memory list *L*. The concrete execution of DCAS algorithm is as follows (depicted in Fig. 1).

Algorithm 1: Asynchronous Snapshot based on Distributed Cache

```
1.  function initial (function, size, input, output)
2.    F ← function;
3.    IQ ← input;
4.    OQ ← output;
5.  function process (R ← IQ )
6.    OQ ← (F, R)
7.  function listener (OO)
8.    L ← L ∪ R;
9.    if L is full then
10.       trigger (OO);
11.       L ← Ø;
12. function trigger (OO)
13.   (S_o,S_u) ←OO;
14.   OF ← L;
15.   snapshot (S_o,S_u,OF);
16. function snapshot (S_o,S_u,OF)
17.   M ← (S_o,S_u,OF) ;
```

1) after the task node starts, every input stream in operators will be assigned to a distributed in-memory list. This list is used to cache the record in snapshot interval. In addition, it will also be allocated a distributed in-memory key-value structure to provide storage of snapshot. 2) As shown in Fig.1-a), the operator computes records from all input streams, and caches them to the distributed in-memory list as a repeated consumption of original

records. 3) As shown in Fig.1-b), when the operator has processed records in a snapshot interval, its listener will trigger a checkpoint to store the state (e.g. operator state, user-defined state and record offset) as key-value pairs to the distributed in-memory key-value structure. At the same time, operator can continue to receive records and cache them. 4) After the completion of the snapshot, it will clear the distributed in-memory list which has been processed completely.

4.3 Distributed Storage Strategy

In DCAS algorithm, we also need to optimize the storage management of snapshot, in order to provide fast and precise recovery. We use in-memory distributed cache to implement the reliable storage and fast recovery.

Snapshot contains state and offset in a snapshot interval. If they are stored in local memory, data loss would occur after task node failure; if they are stored in disk or remote database, the cost of failure recovery time will increase. Therefore, we offer a distributed in-memory key-value structure which is provided by distributed cache to store them. In this way, we can guarantee the reliability of them by the backup mechanism.

5 FAILURE RECOVERY

After restarting failed task node, our failure recovery approach will take the following steps: 1) the distributed in-memory queue, list and key-value structure will regain the record backups from cluster. 2) Task node performs *restore*() method, and then the state of each operator will be initialized to the information in the latest snapshot. Through the offset in the latest snapshot, we can find the first position which operator should replay in the distributed in-memory list, and then deal with them. In addition, the source operation should get source records from Kafka, with the offset stored in distributed in-memory map. 3) Operator continues to compute data.

Fig. 2 shows an example of a failure recovery instance. In the task execution flow, other task nodes in the cluster without failure can still work normally. The failed task node can only replay records in part of distributed in-memory list L with the help of offset *OF*. This offset is stored in distributed in-memory key-value structure *M*. Through (*S_o,S_u*) in M, it can obtain state of *OO* and implement "exactly once" semantic guarantee. Then, they can continue to process R by *process*(*R*). There will be lower recovery latency, due to the failure recovery does not need to replay data from data source and restart the entire processing. Meanwhile, we can provide efficient data access with the help of distributed cache.

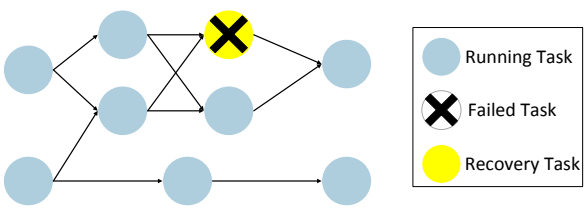

Figure 2. Recovery scheme

Algorithm 2: Failure recovery

```
1. function restore(OO)
2.    (S_o,S_u,OF) ← M;
3.    OO ← S_o,S_u;
4.    R ← (OF+1) in L;
5.    process(R ) ;
6. function process (R )
7.    OQ ← (F, R);
8. function sourceRecover()
9.    OF ← M ;
10.   R ← OF in Kafka ;
11.   IQ ← R ;
```

6 EVALUATION

On the basis of the above implementation, we also validate the ABS algorithm of Flink on the same platform. The goal of this experiment is to contrast and analyze the recovery latency and runtime overhead between DCAS algorithm and ABS algorithm.

6.1 Experiment Setup

Experimental setup includes a plurality of virtual machines (8G in-memory and 4-core CPU), and Gigabit Ethernet network environment. In our experiment, each virtual machine can support a certain number of task nodes. We offer several operators for logical operations, such as map, filter, and aggregation and so on. We send the data source through multi-thread to Kafka message queue, and after the records are executed completely, the final results will be sent to Kafka ultimately.

6.2 Recovery latency analysis

6.2.1 Impact of number of simultaneous failures

We build a cluster with 8 nodes, and use 200 snapshot intervals. During the runtime of system, we stop different number of task nodes, and measure the recovery latency. Fig. 3 shows results averaged over 5 runs for different number of simultaneous failures. By comparing the failure recovery latency between DCAS algorithm and ABS algorithm, we can observe the following:

First, DCAS achieves much faster recovery than ABS in different number of simultaneous failures. Experimental results show that, DCAS algorithm can achieve 6x faster recovery than ABS in the best case and 2x faster recovery even in the worst case. The reasons can be summarized as follows: 1) ABS needs to obtain state from external storage such as HDFS [18] and obtain records from data source like Apache Kafka [14] to replay records. These can generate excessive overhead of disk I/O. Different from ABS, DCAS can obtain state from distributed in-memory key-value structure and records from distributed in-memory list, which can recover rapidly from failure because of the fast speed of distributed cache. 2) When node failure occurs, ABS algorithm needs to restart the whole previous nodes in the execution flow. Especially, when there are amount of operators, or the failed node

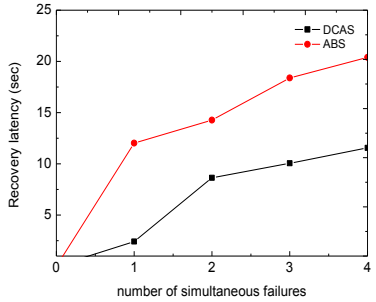

Figure 3. Recovery latency for different number of failures

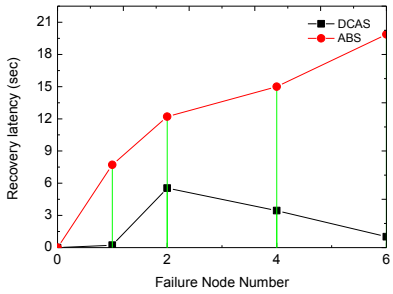

Figure 5. Recovery latency for certain failure node

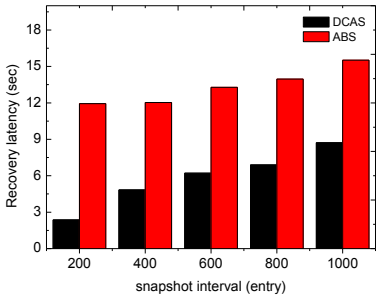

Figure 6. Recovery latency for different snapshot interval

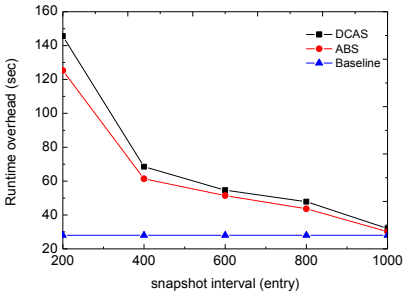

Figure 7. Runtime for different snapshot interval

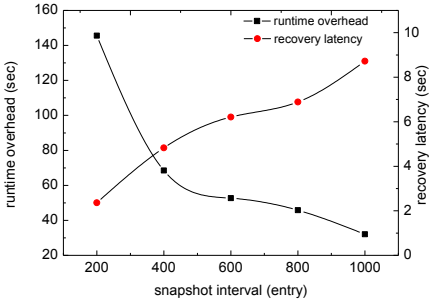

Figure 8. Tradeoff between runtime overhead and recovery latency

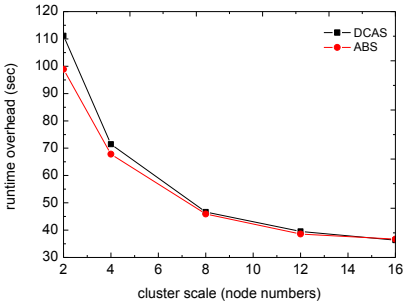

Figure 9. Runtime for different cluster size

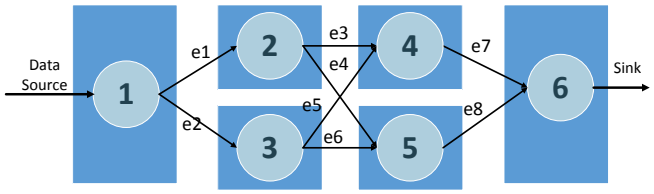

Figure 4. Example of failure recovery

is far from the data source, ABS algorithm has high recovery latency. On the contrary, DCAS just needs to restart the failure node, and other nodes can operate normally.

6.2.2 Impact of certain node failures

We build a cluster with 6 nodes and use 200 snapshot intervals. As shown in Fig. 4, rectangles denote virtual machines (VMs) namely task nodes; the circles represent operators; In addition, the line between two operators represents data stream. We use this execution flow to study the impact of certain node failures.

In Fig. 5, the abscissa represents the mark of failure node, and the ordinate represents recovery time. The experimental results show us that if the failure node is much further away from data source, the recovery latency of ABS algorithm is higher than that in DCAS algorithm. These results are cause by replaying records from data source and restarting all nodes which are in front of the failure node in ABS algorithm. For example, if node 4 failed, the whole nodes from 1 to 4 needs to recalculate records after the latest snapshot, that can obviously increase recovery latency.

Particularly, if the failed node is close to the end of the execution flow, there will be longer process of reproduction. DCAS can only restart the failed node and the other nodes will not be affected. In the above example, DCAS can simply obtain the latest snapshot of the node 4, and recover to normal operator quickly with the help of distributed cache.

6.2.3 Impact of snapshot interval

Fig. 6 shows the recovery latency of DCAS and ABS for different snapshot interval in a cluster with 8 nodes. Recovery latency increases with larger snapshot interval because there are more records should be replayed in a snapshot interval. Of course, DCAS still has lower recovery latency than ABS because of the fast recovery from distributed in-memory structures.

6.3 Runtime overhead analysis

6.3.1 Impact of snapshot interval

We use four virtual machines (VMs) to observe the impact of snapshot interval. Data sources generate ten million records which are firstly sent to Kafka, and then we receive records from Kafka. Fig. 7 shows the runtime impact of DCAS and ABS in different snapshot interval. The lines labeled “DCAS”, “ABS” and “Baseline” present the runtime latency caused by DCAS algorithm, ABS algorithm and none failure recovery mechanism. These performance results can be summarized as follows:

The snapshot interval can affect the performance of system in runtime. The results show that as the snapshot interval is increasing, the runtime overhead decreases rapidly and gradually stabilized to the “Baseline”. The reason for this phenomenon is as follows. In smaller snapshot interval, ABS blocks input stream

frequently to ensure "exactly once" semantic guarantee, while DCAS has to record offset in distributed in-memory list frequently, which also generates more overhead in runtime. In addition, DCAS has a slight high impact on the performance than ABS. DCAS should spend more time in doing backup remotely, so there may be higher network latency in runtime. However, the average impact of runtime overhead in DCAS compared to ABS is no more than 8%.

6.3.2 Tradeoff between recovery and runtime overhead

In Fig. 8, we demonstrate there are tradeoffs between recovery latency and runtime overhead for different snapshot intervals. We can observe from the chart that the larger the snapshot intervals, the lower the impact on records processing, but the recovery latency will be higher. So the snapshot interval should be chosen based on the suitable tradeoff between recovery latency and runtime overhead. When the snapshot interval is near to 400, there is a better balance between recovery and runtime overhead.

6.3.3 Impact of cluster size

In this experiment, the snapshot interval is fixed as 400 which is a good choice for balancing recovery and runtime overhead. Meanwhile, data sources send ten million records to Kafka. We increase cluster size from 2 to 16, and we can find the following conclusions as shown in Fig. 9. First, as the growth of the cluster size, the runtime overhead is reduced gradually. Especially when the cluster size is from 2 to 4, the runtime overhead decreases by 25%. Second, DCAS has no more than 6% runtime overhead than ABS in average and as the growth of cluster size, the gap of runtime overhead between DCAS and ABS is almost close to zero.

7 RELATED WORK

According to our research, the existing stream processing system (SPS) has supported failure recovery method to provide "exactly once" semantic guarantee, such as Apache Spark Streaming [12-13,20], Storm Trident [7], Google Cloud Dataflow [8,15], Apache Flink [9-10] and so on.

A very popular way Chandy and Lamport [17] has proposed asynchronous snapshot and simultaneously do upstream backup long before; and it also presented a global state detection algorithm, which can achieve the consistent storage of state and operator. However, this way still suffers from higher recovery latency. Apache Flink makes some improvements on the basis of the above approach. It uses the checkpoint [16] in neighboring barrier interval to take a snapshot of the global state, and store it in external reliable storage such as HDFS [18] or RocksDB [19]. At the same time, it provides the reliability of state, and replays records using data which is retrieved in Apache Kafka. After the node failed, it needs to retrieve the latest snapshot from the external reliable storage, which will generate excessive disk IO overhead; in addition, taking out records from the data source to restart the entire computing processing can produce high recovery latency when there are a large number of operators.

8 CONCLUSION

We put forward a failure recovery approach with high efficiency, low recovery latency and "exactly once" semantic guarantee. In this recovery approach, we recover records from intermediate results instead of data source; in addition, we propose DCAS algorithm which asynchronously snapshot state. Most importantly, DCAS use in-memory distributed cache to provide records and snapshots backup. Based on the above, SPS can quickly regain intermediate results and lost state from in-memory distributed cache. The experimental results show that this approach achieves 2-6x faster recovery than ABS algorithm in Flink and the runtime overhead is no more than 6% compared to the ABS algorithm.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 61572480; National Key Technology R&D Program of China No. 2015BAF04B02-2; Youth Innovation Promotion Association, CAS under Grant No. 2015088.

REFERENCES

[1] Manyika J, Chui M, Brown B, et al. Big data: The next frontier for innovation, competition, and productivity[J]. 2011.
[2] Kamburugamuve, Supun, et al. Survey of distributed stream processing for large stream sources. Technical report. 2013.
[3] Cui X C, Yu X H, Liu Y, et al. Distributed stream processing: a survey. Journal of computer Research and Development, 2015, 52(2): 318~332.
[4] Hirzel M, Soulé R, Schneider S, et al. A catalog of stream processing optimizations[J]. ACM Computing Surveys (CSUR), 2014, 46(4): 46.
[5] Castro Fernandez R, Migliavacca M, Kalyvianaki E, et al. Integrating scale out and fault tolerance in stream processing using operator state management[C]//Proceedings of the 2013 ACM SIGMOD international conference on Management of data. ACM, 2013: 725-736.
[6] Apache Flink. [Online]. http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/.
[7] Apache Storm. Trident API Overview. [Online]. http://storm.apache.org/documentation/Trident-API-Overview.html.
[8] Perry F. Sneak peek: Google cloud dataflow, a cloud-native data processing service[J]. URL: http://googlecloudplatform. blogspot. com/2014/06/sneak-peek-googlecloud-dataflow-a-cloud-native-dataprocessing-service. html, 2014.
[9] Apache Flink.[online].https://flink.apache.org/.
[10] P. Carbone, G. F´ora, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight Asynchronous Snapshots for Distributed Dataflows," arXiv preprint arXiv:1506.08603, 2015.
[11] Hazelcast.[online].http://docs.hazelcast.org/docs.
[12] Apache spark streaming.[online].http://spark.apache.org/streaming/.
[13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In Symposium on Operating Systems Principles, 2013.
[14] Apache Kafka.[online].http://kafka.apache.org/.
[15] Krishnan S P T, Gonzalez J L U. Google Cloud Dataflow[M]//Building Your Next Big Thing with Google Cloud Platform. Apress, 2015: 255-275.
[16] Zaharia M, Das T, Li H, et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters[C]//Presented as part of the. 2012.
[17] Chandy K M, Lamport L. Distributed snapshots: determining global states of distributed systems[J]. ACM Transactions on Computer Systems (TOCS), 1985, 3(1): 63-75.
[18] HDFS.[online].http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.
[19] RocksDB.[online].http://rocksdb.org.
[20] Spark Streaming. [online]. https://databricks.com/blog/2015/01/15/ improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html.
[21] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 2015.
[22] Khan, Iqbal. "Distributed Caching On The Path To Scalability". MSDN (July 2009). Retrieved 2012-03-30.