World Wide Web (2019) 22:2471–2496
https://doi.org/10.1007/s11280-018-0656-0

Integrating workload balancing and fault tolerance in distributed stream processing system

Junhua Fang[1,4] · Pingfu Chao[2] · Rong Zhang[3] · Xiaofang Zhou[1,2]

Received: 5 April 2018 / Revised: 6 December 2018 / Accepted: 17 December 2018 /
Published online: 7 January 2019

Abstract
Distributed Stream Processing Engine (DSPE) is designed for processing continuous streams so as to achieve the real-time performance with low latency guaranteed. To satisfy such requirement, the availability and efficiency are the main concern of the DSPE system, which can be achieved by a proper design of the fault tolerance module and the workload balancing module, respectively. However, the inherent characteristics of data streams, including persistence, dynamic and unpredictability, pose great challenges in satisfying both properties. As far as we know, most of the state-of-the-art DSPE systems take either fault tolerance or workload balancing as its single optimization goal, which in turn receives a higher resource overhead or longer recovery time. In this paper, we combine the fault tolerance and workload balancing mechanisms in the DSPE to reduce the overall resource consumption while keeping the system interactive, high-throughput, scalable and highly available. Based on our data-level replication strategy, our method can handle the dynamic data skewness and node failure scenario: during the distribution fluctuation of the incoming stream, we rebalance the workload by selectively inactivate the data in high-load nodes and activate their replicas on low-load nodes to minimize the migration overhead within the stateful operator; when a fault occurs in the process, the system activates the replicas of the data affected to ensure the correctness while keeping the workload balanced. Extensive experiments on various join workloads on both benchmark data and real data show our superior performance compared with baseline systems.

Keywords Real-time data processing · Distributed systems · Load Balancing · High availability computing

✉ Junhua Fang
jhfang@suda.edu.cn

[1] Institute of Artificial Intelligence, School of Computer Science and Technology, Soochow University, Suzhou 215006, China

[2] School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, QLD 4072, Australia

[3] School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

[4] Neusoft Corporation, Shenyang 110179, China

1 Introduction

Nowadays, the data are time-sensitive in many areas, which means the "freshness" of the data becomes the deterministic factor affecting their value. The Distributed Stream Processing Engine has been playing an essential role in the real-time applications, such as online detection of the instrument malfunction in telecommunication station controller, quick detection of fraudulence in financial markets and real-time integration of multi-source personal data. To preserve the "freshness" of the input data, such stream processing system requires to be highly efficient in data processing and invulnerable to system faults. However, due to the inherent weaknesses of the distributed system on workload balancing and system faults, the current research on workload balancing and fault tolerance in DSPE has already become an indispensable part in the development of stream processing system.

Nevertheless, achieving both high efficiency and high availability is a challenging task. Hitherto, the existing optimizations can only take either workload balancing or fault tolerance as the single optimization objective. Specifically, the main idea of those solutions are categorized as follows:

Workload balancing Workload skewness and variance are common phenomena in distributed stream processing engines. When massive stream data flood into a distributed system, even a slight distribution change on the incoming data stream may affect the system performance significantly. Traditional load balancing approaches attempt to balance the workload of the system by evenly assigning a variety of heterogeneous tasks to distributed nodes [22, 29]. The existing strategies consist of two classes, namely the operator-based and the data-based strategies. The operator-based strategies [23, 37, 38] assume the basic computation units are operators, while data-based strategies [9, 24, 25] allow the system to repartition the workload according to the keys of the tuples in the stream.

Fault tolerance High availability is a key requirement of the stream processing systems [7] since they process the live data which are continuous, dynamic and unpredictable. In particular, data stream usually does not allow the system to redo a process because the volume of those incoming data may be boundless and the state information cannot be stored completely. Furthermore, since a topology in DSPE is usually distributed over a set of machines for high performance and scalability, the possibility of a node failure increases simultaneously. Two studies [15, 35] estimate the failure probability for an individual cloud host is 4 or 8%, respectively, which means it is very likely to experience failures during runtime. Existing fault-tolerant techniques can be classified into two major classes [16]: active replication and upstream backup. The active replication [4, 13, 17] deploys two or more identical instances of the same operator on different hosts, while upstream backup [6, 18, 28] restores the operator in case of any error based on a checkpoint of the current operator state and the output queue.

To achieve both high-performance and high-availability in one solution, it is straightforward to just combine the workload-balancing and fault-tolerant strategies mentioned above. However, such a simple solution works at the expense of huge resource consumption since the parallel processing tasks in both active and duplication operators have to handle the workload imbalance caused by input data independently.

Meanwhile, the current research trend on both workload balancing and fault tolerance starts to focus on a finer granularity. Specifically, in terms of the workload balancing, the

current key-based methods are more resource-sensitive and flexible in responding to the dynamic data flows compared with operator-based strategies. Regarding the fault tolerance, the backup granularity from big to small can be divided into cluster-based [14, 31], logical operator-based [30, 34] and execution thread-based [19, 32], and it is no doubt that a finer backup granularity makes the system more adaptive to the duplicate contents it wants.

According to the above observations, it seems feasible to have a solution which integrates both workload balancing and key-based fault tolerance. For workload balancing, the degree of the workload imbalance is determined by multiple factors, such as the key cardinality, the degree of data skewness and the number of parallel processing instances [10]. In particular, the degree of workload imbalance is inversely proportional to the key cardinality. For fault-tolerant, the backup granularity determines the flexibility of the fault-tolerant strategy. In other words, data-based backup strategy is a more flexible way to ensure the system fault tolerance while providing more room for utilizing available resources and a lower recovery latency. Based on the above, the advantages of such combination can be described as: 1) the replicas in integrated architecture enlarge the key cardinality, which means the number of possible balance plan increases significantly because more keys are movable; 2) a finer-grained fault-tolerant strategy enables the system to backup their state information in a more flexible manner according to the priority their incoming data; 3) the parallel backup based on data-level can significantly reduces the recovery latency.

Nevertheless, a fine-grained fault tolerance also raises new challenges to the distributed system in regard to handling the dynamic of data stream while maintaining high resource utilization rate at any time. In our solution, to achieve a data-level fault-tolerant mechanism, we have to tackle the following questions:

Q1) How to distribute the active and backup data to ensure the system correctness during the processing stage or after the task failure occurs; **Q2)** When the workload among parallel processing task instances is imbalanced, how to generate a lightweight computing model to repartition states among parallel tasks so that the workload is rebalanced; **Q3)** How to create migration plans for a set of involved tasks as quickly as possible especially for computationally heavy tasks with limited migration capacity.

In this paper, we propose a fine-grained fault-tolerant strategy for economical resource utilization on the premise of efficient data recovery. By considering both workload balancing and fault tolerance, our proposed method can achieve low recovery latency with efficient balance processing. In the following sections, we will introduce our work by answering **Q1)** in Sections 3 and 4.1, **Q2)** in Sections 4.2.1 and 4.2.2, and **Q3)** in Section 4.2.3,. Overall, the technical contributions of this paper include:

1) We design a fault-tolerant framework that combines workload balancing at the same time, which make the workload-balancing and fault-tolerant mechanisms "mutually benefit and win-win" to each other including conducive to the generation of rebalancing strategies, reduction of migration costs and more flexible backup strategies.

2) We propose a rebalance approach that takes into account the fault-tolerant mechanism. In the premise of workload balance, we select the primary and secondary data storage locations in a location-sensitive manner and selectively activate the backup data for fault tolerance to reduce the migration cost.

3) We present a concise data structure for identifying activities and backup actions, and design a data routing solution to ensure the correctness of the system's operations, while making the system scalable, effective and efficient.

4) We implement our algorithms on Apache Storm and compare our proposed techniques with existing methods by extensive experiments on abundant synthetic and real datasets.

The remainder of this paper is organized as follows: Section 2 describes the background and problem formulation. Section 3 presents how data-level fault-tolerant method guarantees the system correctness. Section 4 proposes algorithms to achieve the optimization goal of our paper. Section 5 presents empirical evaluations of our proposal. Section 6 reviews a wide spectrum of related studies on stream join and workload balancing in distributed systems. Section 7 finally concludes the paper and addresses future research directions.

2 Preliminaries

2.1 Distributed stream processing engine

A Distributed Stream Processing Engine, such as the topology in Apache Storm, can be represented as a directed graph, whose vertex is an operator executing user-defined functions, and each edge between two vertices represents a data stream(a collection of key-value pairs called *tuples*) produced by an upstream operator and consumed by a downstream operator. To maximize the throughput of the stream processing system and improve the utilization rate of the computation resource, the workload of a logical operator is usually partitioned and concurrently processed by a number of threads, known as *tasks*.

The workload partitioning among concrete tasks is modelled as a mapping from key domain to the running tasks in the succeeding operator. Intuitively, the workload partitioning can be done by a simple mapping function (e.g., consistent hashing), which assigns the tasks with the same key to a fixed node consistently. As discussed in the previous section, despite the huge advantages of hashing on memory and computation efficiency, such solution is vulnerable to workload variance and key skewness. Another solution is to explicitly assign the tuples based on a carefully optimized routing table, which specifies the destination of the tuples by a map structure on the keys. Although such approach is more flexible on dynamic workload repartitioning, it is impractical due to its high operational cost on both memory and computation.

In our model, the time domain is discretized into consecutive time intervals, denoted by $(T_1, T_2, \ldots, T_i, \ldots)$. At the i-th interval, given a pair of connected operators in the processing logic, U and D, we use $\mathcal{U}$ and $\mathcal{D}$ to denote the set of task instances within the upstream and downstream operators, respectively, and use $N_U = |\mathcal{U}|$ and $N_D = |\mathcal{D}|$ to represent the corresponding numbers of task instances. A tuple is denoted by $\tau = (k, v)$, in which k is the key of the tuple in the key domain $\mathcal{K}$ and v is the corresponding value. We assume N_U and N_D are predefined without immediate change. The discussion on the dynamic resource rescheduling, i.e. changing N_U and N_D, is out of the scope of this paper since it involves global optimization based on the dependency analysis on the whole processing logic, e.g. [11]. All notations used in the rest of the paper are summarized in Table 1.

2.2 Problem conclusion

Before any tuple is emitted by one of the upstream operator instances, a Tuple-to-Instance Assignment Function(*TIAF*) should be predefined to determine its destination instance in the downstream operator. Although functions, like *random* and *round-robin*, are inherently load-balanced among all the instances of the downstream operator, it is hardly applicable to the stateful operators. In contrary, the key-based *TIAF* such as *hashing* guarantees the computational correctness for stateful operators, while the load balancing among instances is vulnerable to the skewness of the key distribution. Since the key distribution usually follows

Table 1 Table of notations

Notations	Description
T_i	i-th time interval
(k, v)	a key-value pair in the data stream
k	key of the tuple
v	value of the tuple
d	task in downstream operator
$\mathcal{R}$	the routing table available to U
$c_i(k)$	Computation cost of tuples with k in T_i
$g_i(k)$	frequency of key k in time interval T_i
$L_i(d, F)$	the total workload of d under assignment function F
$\bar{L}_i$	average load of all instances in U in T_i
$\theta_i(d, F)$	the load balance factor of task instance d
$\theta_{\max}$	Upper bound of imbalance tolerance
$S_i(k, w)$	the memory cost of key k with w time intervals at T_i
$M_i(w, F_i, F_{i-1})$	the total migration cost by replacing F_i with F_{i-1} at time interval T_i

a long-tail distribution, i.e. a small number of keys have a huge number of occurrences, a key-based function will lead to the load imbalance, resource inefficiency and performance degradation. Furthermore, to apply the key-based fault tolerance to the system, the routing strategy for backup data is also an essential component in *TIAF*.

The goal of our paper is to combine the fault-tolerant and workload-balancing mechanisms in the DSPE to reduce the overall resource consumption while achieving low-latency, high throughput, scalability and high availability in DSPE. In our system, the *TIAF* is, essentially, responsible for routing the incoming tuples. It is able to adjust the workload of parallel processing tasks through redirecting incoming tuple, ensure the correctness of data processing through switching the active state between active tuples and their replicas. Based on this, we develop our *TIAF* based on a new mixed routing strategy, which combines a hash-based randomized strategy(explicit routing) and a key-based routing strategy(implicit routing), as shown in Figure 1. Our *TIAF* is a self-adaptive and flexible module that rebalances the uneven loads among instances and recovers the data state with the minimum possible costs of key state migration.

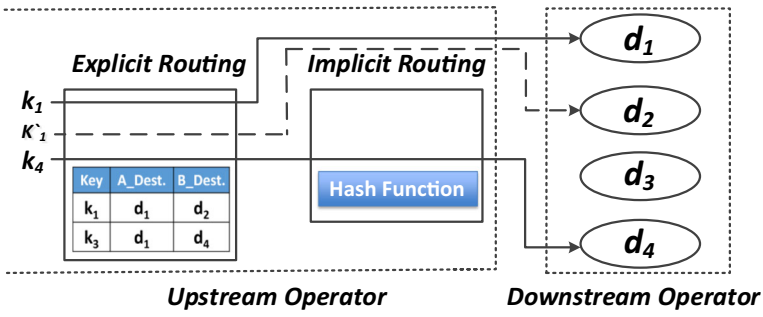


Figure 1 Work flow of the new mixed routing with a small routing table and hash function

As shown in Figure 1, a routing table is maintained by the system, named Explicit Routing, but only for a handful of keys. When a new output tuple(k_1) and its replicas(k_1') are generated for the downstream operator, the upstream operator first checks if the key exists in the routing table. If a valid entry is found in the table, the tuple is transmitted to the target task instance with the matching id. Otherwise, a hash function is employed to generate the deterministic task instance id by randomized hashing. The implicit routing is responsible for the intact key's routing, and is usually a function mapping by default, like for an active key, the mapping function to its destination is $k\%n$, while the mapping of its backup key can be $(k+1)\%n$.

Although this scheme with routing table achieves high reliability by easily redirecting the keys to new worker threads with simple editing on the routing table, the routing table which generated by the workload balancing adjustment will grow boundlessly if no size limit applied. Obviously, a huge routing table will increase the cost on both memory and computation because each task instance must hold the same routing table and every incoming tuple must first access it to see if its key is covered. Hence, in our paper, by carefully maintaining the routing table to satisfy the maximal size constraint, we limit the memory consumption and computation cost to an acceptable amount, while the flexibility and effectiveness are still preserved by appropriately updating the routing table based on the evolving workload with the output tuple stream.

The following discussion on the key-based *TIAF* design will focus on the individual edge connecting a pair of upstream and downstream operators. Since our approach can easily be applied to all the edges simultaneously and independently, the resource utilization and overall performance of the whole DSPE is optimized.

Problems that are solved by our method are summarized as follows:

Correctness guarantee A key-based workload distribution mechanism works as a mapping $F : \mathcal{K} \rightarrow \mathcal{D}$, such that a tuple (k, v) is sent to task instance $F(k)$ by evaluating the key k with the function F. Without loss of generality, we assume a universal hash function $h : \mathcal{K} \rightarrow \mathcal{D}$ is available to the system for general key assignment. A typical hash function implementation is consistent hashing, which is believed to be the best option for a distributed system with dynamic parallelism on N_D. A routing table $\mathcal{R}$ of size N_R is a group of pairs from $\mathcal{K} \times \mathcal{D}$, specifying the destination task instances for keys in $\mathcal{R}$. The mixed routing strategy shown in Figure 1 is thus modelled by the following equation:

$$F(k) = \begin{cases} (A.d, \{B.d\}) & \text{if } \exists\ (k, d) \in \mathcal{R}, \\ (h(k), \{f(h(k))\}), & \text{otherwise.} \end{cases} \tag{1}$$

where $A.d$ is the destination task of the active keys, while $B.d$ represents the destination(s) for backup keys. Similarly, $h(k)$ and $f(h(k))$ are the mapping function for active and backup keys, respectively. Note that the destination of the backup keys is a task set since the importance of each key varies, which can be obtained according to a "descriptor" [5]. The "descriptor" is a document summarizing, with a set of concise attributes, the computational behaviour of tasks and the expected characteristics of application input streams.

According to the (1), workload redistribution is enabled by editing the routing table $\mathcal{R}$ with assignment function F. In the following, we provide formal analysis on the general properties of the assignment function $F(\cdot)$. Different from the hash mapping function, each entry in the routing table $\mathcal{R}$ incurs extra overheads in terms of the memory space and time consumptions. Without loss of generality, the size of routing table can be estimated by $R = \sum_{r \in \mathcal{R}} (r.k + r.A.d + r.B.d)$, which becomes a measure of the total cost of this table.

Load balance We use $g_i(k)$ to denote the occurrence count of tuples with key k during the ith interval, and define the computation cost $c_i(k)$ for processing tuples with k to be an monotonously increasing function to its occurrence count, i.e. $c_i(k) = f(g_i(k))$, where $f(\bullet) > 0$, and the rationale for the increasing assumption is axiomatic since a key with more occurrence count causes more computation cost. Here, we adopt $c_i(k) = g_i(k)^\alpha, \alpha > 0$ to be the cost function throughout the paper for easier illustration. In fact, our approach and analysis are by no means restricted to any specific form. For each downstream operator instance $d \in \mathcal{D}_i$, its total workload during the i^{th} interval, denoted by $L_i(d, F_i)$, is the total computation cost of all the keys assigned to it:

$$L_i(d, F) = \sum_{\{k|F(k)=d, k\in\mathcal{K}\}} c_i(k). \tag{2}$$

Since the load balancing among task instances of the downstream operator D is one of the main goals of our proposal in this paper, we define the balance indicator $\theta_i(d, F)$ for each of the instances to characterize the degree of load imbalance:

$$\theta_i(d, F) = \frac{\left|L_i(d, F) - \bar{L}_i\right|}{\bar{L}_i}, \tag{3}$$

where $\bar{L}_i = \frac{1}{N_D} \sum_{d\in\mathcal{D}} L_i(d, F)$ is the average load of all task instances in $\mathcal{D}$. Actually, it is usually impossible to achieve absolute load balancing with $\theta_i(d, F) = 0$ for every task instance d. In practice, a lower bound $\theta_{\min}$ and an upper bound $\theta_{\max}$ are specified by the system administrator, such that the workload of task instance d is approximately balanced if $\theta_{\min} \le \theta_i(d, F) \le \theta_{\max}$.

Lightweight adjustment Keep rebalancing the workload with minimal migration overhead for the stateful operator can help to make the system performs efficiently in a consistent way. However, due to the updates on $F_i(k)$ after each load-balancing step, there are changes on key assignment results in the i^{th} interval comparing to the $(i-1)^{th}$, i.e. $\exists k$, s.t. $F_i(k) \neq F_{i-1}(k)$. the states of these affected keys must be migrated to the newly assigned instance to ensure the correctness of processing results. Hence, we use $\mathcal{K}_i \subseteq \mathcal{K}$ to denote this subset of affected keys:

$$\mathcal{K}_i = \{k|F_i(k) \neq F_{i-1}(k), k \in \mathcal{K}\}. \tag{4}$$

The key state migration includes all the historical states within the given window w. Thus, the total migration cost, denoted by $M_i\ (w, F_i, F_{i-1})$, can be defined as:

$$M_i\ (w, F_i, F_{i-1}) = \sum_{k\in\mathcal{K}_i} S_i(k, w). \tag{5}$$

Note that besides the parameter w, $M_i\ (w, F_i, F_{i-1})$ is, although not explicitly expressed, also affected by the two assignment functions F_{i-1} and F_i, which results in the affected key set $\mathcal{K}_i$.

Based on the model mentioned in correctness guarantee, workload balance and a lightweight adjustment action, we now define our dynamic workload distribution problem, with the objectives on (i) ensuring the correctness of tuple routing and maintaining an acceptable routing table size; (ii) balancing workload among all the downstream instances; and (iii) providing a lightweight adjustment action during the processing architecture transformation.

3 Fault-tolerance mechanism

In this section, we first introduce how data-level fault-tolerant method guarantees the system correctness. Then we discuss the consistency issue in failure recovery and demonstrate our detailed fault-tolerant process.

3.1 Synchronization protocols

The system availability is determined by two factors: the synchronization process between the active and backup operations, and the recovery process after task failure. Note that, the fault-tolerant strategy may differ according to the requirements from different operations and applications. For example, the *filter* and *map* operations usually deal with single data primitive, so the process of those operations are not affected by other procedures, while the operations like *aggregation*, *join* and *union* are usually applied on multiple data sources, which requires the timestamps of different data source to follow the same sequence. To further understand the challenges of our solution, we now discuss the potential consequences of task failure to the system availability and consistency. Our discussion starts with both operation and data aspects. In terms of the time dependency of system output, the current operations can be categorized into two types: time-dependent and time-independent operations.

Definition 1 In data stream processing, an operation is called **time-dependent** operation if the process of the operation relies on the timestamps of the input data.

Definition 2 In data stream processing, an operation is called **time-independent** operation if the process of the operation does not rely on the timestamps of the input data.

The different time-dependency feature requires different synchronization level between active/backup data: If the operation is time-dependent, the output result is strongly related to the time feature. In contrary, if the operation is time-independent, the system only needs to ensure the calculation is correct and the data order is the same for both main and backup tasks. Based on this, we define three synchronization protocols which can ensure the correctness of different operations.

The timestamps feature is indispensable in every stream data tuple. For some operators, if the data comes in a muddled order, the processing result of the data sequence may vary from the actual result when the data comes successively. For instance, in the *join* operation, if the disorder happens in either of the data sources, the join results may lose some of the records [20]. Hence, we define the *data order synchronization* based on the data features.

Definition 3 For the data stream input, regardless single-sourced or multi-sourced, if the tuples are increasingly input into active/backup task according to their timestamps, such stream sequence is termed as **data order guaranteed**.

If the active task fails, the system should activate its replica to ensure the output correctness. However, the synchronization between active and backup data usually determines the system availability and consistency [4, 34], i.e. strong synchronization reduces the system availability, while high availability will challenge the synchronization strategy. The following definitions describe the synchronization relationship between active and backup data during different processing stages:

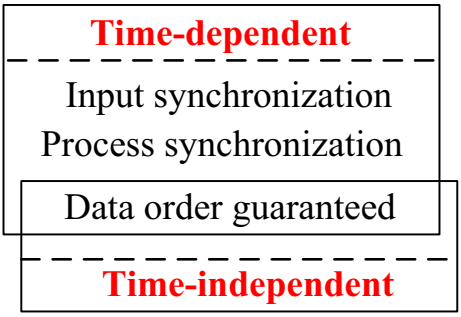


Figure 2 The critical factors for Fault-Tolerance methods under different applications

Definition 4 In the stream data processing system, if the speed of the data input flows to active and backup operations are strongly synchronized, we call the main and backup data input are **input synchronization**.

Definition 5 In the stream data processing system, if the operating progress between active and backup operations are strongly synchronized, we call the main and backup processing task are **process synchronization**.

It is the prerequisite of process synchronization that the active/backup data are sent to processing tasks simultaneously. Meanwhile, the process synchronization is also the precondition of the downstream input synchronization. As for the aforementioned definitions, the time-dependent/independent are used to categorize the operations, the data order guaranteed is to regulate the data input, while the input/process synchronization is to formalize the consistency issue during the active/backup data processing. In simple terms, as shown in Figure 2, time-dependent applications require all the criteria of input/process synchronization and data order guaranteed while time-independent applications only need tuples in active and duplication has the same order to produce the correct results. Based on this, in the rest of this section, we describe our strategies including how to ensure input/process synchronization in Section 3.1.1 and data order in Section 3.1.2.

3.1.1 Input/process synchronization

The output results of the time-dependent operation are strongly related to the time feature(like real-time hot topics), in another word, the result will become invalid if it is processed later than the given time. Hence, making sure data flow rate and processing speed are all synchronized is the only way to ensure the active/backup data can be switched with no delayed. However, it is unrealistic to make sure the active and duplication states of each tuple are strong synchronized both in input and processing procedure, which will incur significant synchronization cost for massive steam processing.

To avoid the expensive synchronization cost caused by continuous synchronization process, as shown in Figure 3, we introduce a reconstruction mechanism for each computation state. This is similar to TimeStream [28], which tracks the state and output dependencies for each vertex during the stream computation. In our computational state reconstruction mechanism, we take each tuple as the state reconstruction granularity and the dependency duplication tuples are maintained according the real application. For instance, in Figure 3, the output result O_{t_i} depends on tuple with timestamp t_i and t_{i-1} when window size is 2.

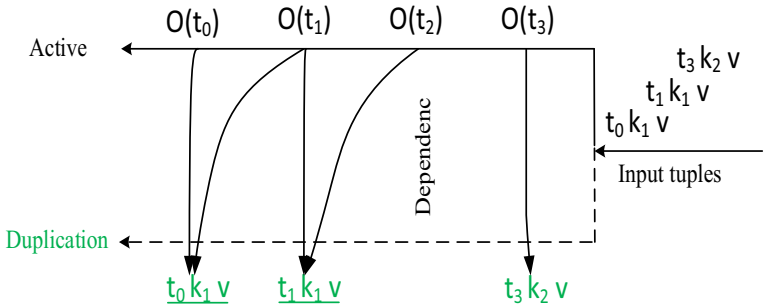


Figure 3 Example of dependency relationship between active and duplication data stream with $w = 2$

The example in Figure 4 shows the recovery flow based on dependencies. Specifically, (t_x, k_y, V) represents a tuple that carries key k_y at time x, and V represents the rest information. In Figure 4, the order of the data input is: tuples carrying k_1 and k_2 are input at time t_0, a tuple with k_1 comes at t_1, t_2 has a tuple with k_2, a tuple k_2 comes at t_3, a tuple k_1 comes at t_5 .... The topology demonstrates two operations: the counting (Count) and the aggregation (TopN). The parallel degree of the counting operation is 2, hence, the k_1 is at active state in d_1 and at duplication state in d_2. In contrary, k_2 is at active state in d_2 and at backup state in d_1. Assuming that the physical server containing task d_1 fails between t_5 and t_6, at that moment, the counting of k_1 in d_1 has been processed till t_5, and the counting result has been sent to the downstream task d_3 for aggregation. Meanwhile, due to the network latency, the backup state has not yet processed the tuple (t_5, k_1, V), resulting that the counting result only contains two tuples. Corresponding to the above analysis (Figure 2), if the TopN result is output in real-time (i.e. the result of every operation is strictly time-dependent), either input disordered, input delay or processing delay will cause the results

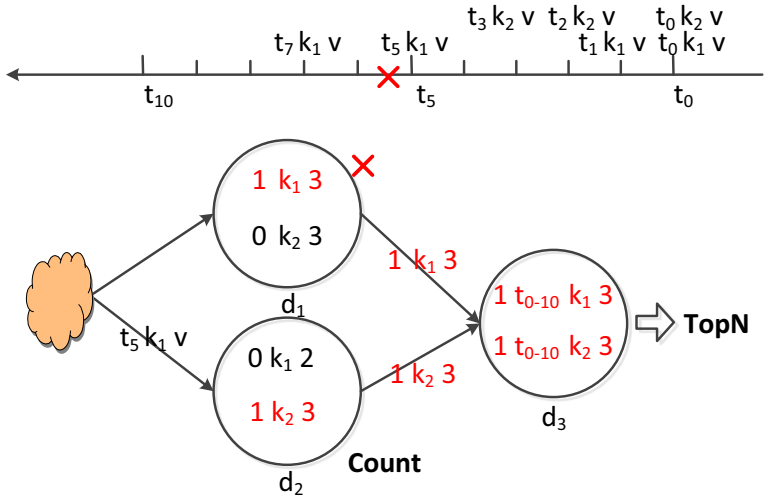


Figure 4 Example of recovery based on dependencies

from main and backup data useless. However, if the data in Figure 4 is time-independent, i.e. the result is not necessarily synchronized with the timestamps(such as log data analysis), the data stream only need to ensure its data is monotonically increasing by the timestamps, and the synchronization of input and processing is not necessary. The Figure 4 shows the case that the backup data is later than the active ones, which can be solved by simply preserve the state of its downstream operation(d_3). If the active data is later, we replace the state of downstream operation with the backup data state.

3.1.2 Data order guarantee

To solve the inputting tuples synchronization issue shown in Figure 2, in this section, we introduce the operation-level data state synchronization solution, which is to ensure synchronization happens before each operation. In the fault-tolerance method proposed in [4], since each copy can provide service individually, the synchronization between copies should be strictly processed to ensure the consistency. Different from the method in [4], to ensure the consistency between main and backup data in terms of the input order, flow rate and processing speed, our data-level fault-tolerance method only sends active keys to the downstream tasks before the data are input into the system. We use a fix-sized cache to store and sort the input data sequence to negate its uncertainty. The Figure 5 shows three different stream data input which is incrementally sorted by time in the cache. The cache size is determined by the quality of data source. A larger cache will reduce the efficiency of data process, while a smaller cache cannot handle an extremely disordered data source.

The buffer strategy shown in Figure 5 guarantees the data order defined in Definition 3 and, meanwhile, ensure the recovery speed after the failure of a processing task. The detailed solution is to set up a buffer for each operation, as denoted by Figure 5, each data sources ($S_{1\sim3}$) in the buffer is sent from one upstream operation. Based on this buffer strategy, the system can be recovered by simply continue the process or replace the fault states, when the input or process inconsistency happens between main and backup tasks. Specifically, if the backup task is slower than the active task in terms of the processing state, the recovery starts from the current state of the downstream task and continue updating it. If the backup task is faster, the recovery can be done by replacing the state in the active task with the state in the backup task. Now we are ready to discuss the recovery process after the task failure.

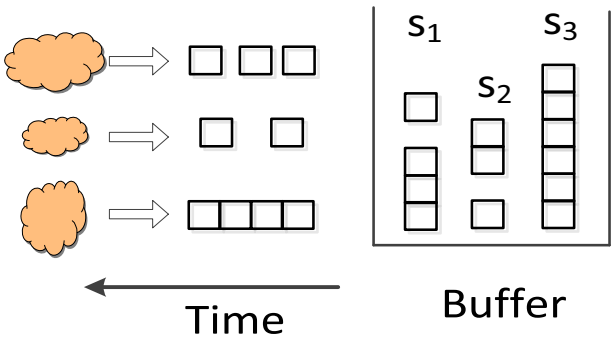


Figure 5 Example of Data Stream inputting

3.2 Fault recovery

As mentioned above, the data stream is a sequence of tuples formatted as $(t, k, v_1, v_2, \ldots)$ [2], t and k are the timestamps and key respectively, while $v \in V$ represent the attributes carried by the tuple. For better understanding, we define a few additional symbols to better represent the state of a tuple, denoted as:

$$(t, P_k, v_v, RN, AC, V^-)$$

In this representation, t is still the timestamps. P_k is the identifier used for tuple distribution, for example, P_k is the hashing result if the tuples are distributed by a hash function. v_v is the value indicator representing how valuable this tuple is, which is specified according to the user's description. RN (Replica Number) indicates that the number of replicas this tuple should have, which is determined by its value v_v. AC is a boolean value indicating its state, $AC = 1$ means this tuple is at active state and its processing results should be sent to two downstream tasks as active and backup data respectively, and $AC = 0$ indicates that it is backup data which need no output action after being processed. Lastly, V^- is the actual information of this tuple.

The Figure 6 shows the process of the main and backup data. For simplicity, we use the key itself to represent the tuples in Figure 6. The keys in red or green are tuples with $AC = 1$ while the black ones are keys carrying $AC = 0$. All six different keys have one replica ($RN = 2$), both the active and backup one are sent to corresponding tasks by the same hash function. In accordance with the aforementioned representation, the tuple k_1 in Figure 6 is represented as $(t, P_{k_1}, v_v, 2, 1, V^-)$ and $k_1^{'}$ is represented as $(t, P_{k_1'}, v_v, 2, 0, V^-)$. The identifiers P_{k_1} and $P_{k_1'}$ will lead the tuples to different tasks, which will be discussed in Section 4.1. In the result output stage, only the red tuples are output to the downstream tasks, the output results include active(marked as green italic) and backup (marked as black italic) data. The routing and backup strategy are the same in the downstream tasks.

When the task failure happens, the data-level fault-tolerance method can recover the data through the following three steps:

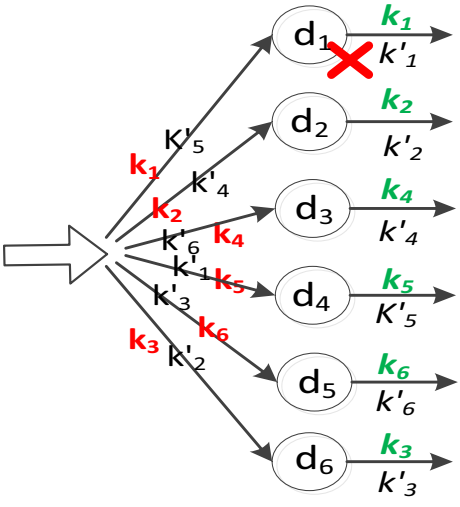


Figure 6 Data-level fault-tolerance work flow

Step-1: Recovery process triggered. When the task failure happens, the system sends the information to all the active keys on the failed task to the tasks containing their backups and trigger the recovery process on those tasks.

Step-2: State confirmation. Since the task failure happens randomly, when the task fails, we cannot get the states of the keys directly from the failed task. Hence, this step ensures that we can obtain the keys' last states through the state of its downstream operations in order to recover the lost data accordingly.

Step-3: Recovery action. With the help of the buffered data of each operation and the state of the lost data obtained from downstream tasks (mentioned in step 2), we now can restore the active data from the backup one.

We continue use Figure 6 as an example, if the task d_1 fails, the system first decides which active key are affected according to the key distribution function, then it sends the *recover* command to the tasks that contains the backup of those lost active keys, i.e. sending recover command to d_4 which contains the backup of k_1. Lastly, the system obtains the downstream operation state of k_1 and the buffer data of the operation in d_4, which can be used to reproduce the state of k_1 when d_1 fails using the state of k_1' in d_4.

Overall, this section introduces the procedure of data-level fault-tolerance method: the system correctness is guaranteed by the strong processing order and the active/backup synchronization which happens in the time-dependent operations. The fault-tolerance method ensures the input order by sorting the input in the buffers. In terms of the data distribution, this method only sends the active data to the downstream main and backup tasks so as to localize the synchronization process. However, due to the occurrence of the replicas, the data routing rule and workload balancing plan have to consider both the active and backup data. Different from the key-based workload balancing method mentioned in [10], this section focus on combining both the fault tolerance and the load balancing. No matter how data are distributed or migrated, the active and backup data are not allowed to be sent to the same processing task. We will introduce the data routing and balancing problem in the next section.

4 Algorithms

Regarding the goal of our paper, $(\theta_i(d, F) \leq \theta_{\max})$ implies the constraint of workload balancing is a variance of the well-known bin packing problem, i.e. objects (keys) must be packed into a fixed number of bins in the way that load imbalance is minimized, which is known to be an NP-hard problem. In this section, we first introduce the routing strategy for the incoming tuples while ensuring its active and duplication state correctness. Then we propose an approximate algorithm based on the least-load first principle. Finally, the analysis of the algorithms is presented to illustrate the theoretical properties of our approaches.

4.1 Tuple routing strategy

As described in Section 2, a tuple τ can be formatted as (k, v), while k is used to determine the destination(s) for τ. According to our mixed routing strategy in (1), incoming tuples are assigned to the processing tasks based on a routing table and a mapping function. Furthermore, for the duplication state of tuples, a Service Level Agreement (SAL) is used to determine how many copies should be generated for different keys. Specifically, SAL is a document containing a set of keys and their importance level. The pseudo code of tuple routing in our mixed architecture is listed in Algorithm 1. It produces the routing

plan for incoming tuple set, which assigns the incoming tuples to one of the instances of the downstream operator and ensures the correctness of the stream processing.

Algorithm 1 first extracts the key form incoming tuple as shown in line 1. Then, it determines the number of replicas for the tuple according to SLA. Finally, it directs the incoming tuple according to the mixed routing strategy in (1). Specifically, if the incoming tuple is to be routed by routing table, which means its key has been migrated, it will be set as active state and routed according to the active destination column in the routing table(line 4–5). At the same time, replicated tuple(s) will be marked as duplication state and sent to task(s) according to the duplication destination column of the routing table(line 6–7). In contrast, the tuple routed by mapping function has the similar procedure except that the routing strategy in line 10 and 12 is determined by the mapping function.

Algorithm 1 Pseudo code of tuple routing.

Input: Input tuple:τ, Service-level agreements: SLA
Output: Active task ID: d, Duplication task ID(set): d'
1 $k \leftarrow \mathcal{N}(\tau) \triangleleft$ extract the key of incoming tuple τ;
2 $\tau' \leftarrow SLA(\tau) \triangleleft$ determine the duplication tuple(s) τ' according to SLA;
3 **if** $k \in \mathcal{R}$ **then**
4 $\quad \tau \leftarrow Set(\tau_{ac=1}) \triangleleft$ set this tuple as active state;
5 $\quad d \leftarrow A.d; \triangleleft$ determine routing for τ according to (1);
6 $\quad \tau' \leftarrow Set(\tau'_{ac=0}) \triangleleft$ set this tuple(s) as duplication state;
7 $\quad d' \leftarrow \{B.d\}$; determine routing for τ according to (1);
8 **else**
9 $\quad \tau \leftarrow Set(\tau_{ac=1})$;
10 $\quad d \leftarrow h(k)$;
11 $\quad \tau' \leftarrow Set(\tau'_{ac=0})$;
12 $\quad d' \leftarrow \{f(h(k))\}$;
13 Return d, d';

4.2 Loadbalance adjustment plan

In this section, we first briefly introduce a basic and heuristic assignment procedure proposed by us, namely *least-load fit decreasing* (LLFD), which also used in our work of [10]. It is inspired by the classic First Fit Decreasing (FFD) designed for the bin packing problem [8]. Then, we propose $FTLB$ (Fault Tolerance and Load Balancing) algorithm for the propose of our paper, to make the system achieving the workload balancing and fault tolerance at the same time.

4.2.1 Least-load fit decreasing (LLFD)

LLFD produces the key assignment plan from an input candidate key set to one of the instances of the downstream operator for the purpose of achieving load balance. The pseudo code of LLFD is listed in Algorithm 2. It first sorts the keys from the given candidate set in descending order of their computation cost $c(k) = g(k)^{\alpha}$ (Line 3). Next, it iteratively assigns each of the keys following this order to the instance that has the least workload so

far (Lines 4–7). If this key-to-instance pair mismatches the result generated by consistent hashing $h(k)$, an entry (k, d) is then added to the explicit assignment table $\mathcal{R}$ (Lines 5–7). Afterwards, it updates the new workload for the affected instance (Line 8) and removes the assigned key from the candidate set (Line 9).

Algorithm 2 Pseudo code of LLFD.

Input: Candidate key set $\mathcal{C}$, instance set $\mathcal{D}$, cons. hash $h(k)$
Output: explicit assignment table $\mathcal{R}$

```
1 Initialize: R = ∅;
2 while C ≠ ∅ do
3     get k with largest computation cost c(k) from C;
4     get d with least load L(d) of all the instances in D;
5     if h(k) ≠ d then
6         add (k, d) to R;
7     L(d) = L(d) + c(k);
8     remove k from C;
```

Assuming if at the beginning of each interval we apply LLFD with all the keys as its input (the candidate key set), this may result in a huge explicit assignment table that contains entries for almost every key. Despite the fact that it is not practical in that it only deals with the load balancing constraints, disregarding either the limitation on assignment table size or the migration cost, it is a base of other practical approaches we will introduce later. Also, we give a proof that the degree of load imbalance among downstream instances produced by $LLFD$ approach is upper bounded, under several types of heavy-tail distributions on key frequencies, e.g. Zipf, etc.

The following theorem shows the robustness and soundness of LLFD on basic load balancing problem. A detailed proof can be found in [10].

Theorem 1 *If there exists a solution for perfect load balancing, i.e.,* $L(d) = \bar{L}, \forall d \in \mathcal{D}$, *LLFD is capable of finding a solution resulting in balancing indicator* $\theta(d, F)$ *no worse than* $\frac{1}{3}(1 - \frac{1}{N_D})$ *for any task instance* d.

Furthermore, the number of keys is more than the number of instances ($|\mathcal{C}| \gg |\mathcal{D}|$) and the accumulative load for the tails in skew data distribution is significant. Therefore, we believe that $LLFD$ can produce a well-balanced adjustment.

4.2.2 Fault tolerance and load balance (FTLB) algorithm

The $LLFD$ reshuffles all the keys according to their workload information. However, reshuffle operation will cause an amazing cost of network transmission and generate a huge assignment table. We have to avoid the reshuffle operation while still preserve the load balancing with our predefined imbalance tolerance under limited assignment table. Here we propose an improved method based on $LLFD$, named as $FTLB$.

The key difference from $FTLB$ to $LLFD$ is that for the i^{th} interval, only a subset of keys are chosen as the candidates for migration instead of the whole key set used by $LLFD$. The balancing procedure consists of two parts: select candidate keys from instances and put candidates to instances.

Algorithm 3 Pseudo code of $FTLB$.

Input: instance set $\mathcal{D}$, hash $h(k)$, Upbound upb, metric $M(k)$
Output: explicit assignment table $\mathcal{R}$

1 Initialize:$\mathcal{C} = \emptyset$, $cursor[] = \infty$;
2 /*1^{st}: select loads into $\mathcal{C}$ from tasks*/;
3 **for** *each* $d \in \mathcal{D}_i$ **do**
4 **while** $L_i(d) > upb$ **do**
5 get k with the best $M(k)$ from set $\{k|F_{i-1}(k) = d\}$;
6 add k to $\mathcal{C}$;
7 $L_i(d) = L_i(d) - c_i(k)$;
8 /*2^{st}: re-distribute loads in $\mathcal{C}$ to tasks*/;
9 **while** $\mathcal{C} \neq \emptyset$ **do**
10 $c(k) \leftarrow$ get k with largest computation cost;
11 $L(d) \leftarrow$ get d with least load of all instances in $\mathcal{D}$;
12 $L(d) = L(d) + c(k)$;
13 **if** $L(d) > upb$ **then**
14 $cursor[d] \leftarrow c(k)$;
15 $\mathcal{C} \leftarrow$ get k with best $M(k)$ and $c(k) < cursor[d]$;
16 update $L(d)$;
17 **if** $L(d) > upb$ **then**
18 undo Line 14–19;
19 get the next least loaded instance d;
20 redo from Line 14;
21 **if** $h(k) \neq d$ **then**
22 add (k, d) to $\mathcal{R}$;
23 remove k from $\mathcal{C}$;
24 Return $\mathcal{R}_i$;

In the first phase, we define evaluation metric $M(k)$ to evaluate the importance of key k for migration purpose and order the keys by their importance. The value of metric $M(k)$ can be obtained according to two factors: the granularity, memory consumption the duplication distribution of keys. Different to the *migration priority index* defined in [10], $M(k)$ takes the duplication information into account. Specifically, the candidate move keys are sorted by an ascending order of the *migration priority index*, which is our evaluation metric $M(k)$ defined by $\gamma_i(k, w) = \frac{S_i(k,w)}{c_i(k)^\beta}$. The physical meaning of the index $\gamma_i(k, w)$ is intuitive, i.e. the key with smaller state size (equivalently migration cost) or larger workload has the higher priority to be migrated, and the parameter β is used for tuning the weights between the two factors. The *migration priority index* here can be defined as $M(k) = \frac{S_i(k,w)}{c_i(k)^\beta} \cdot dn_i(k)^\gamma$ where $dn_i(k)$ is the number of duplication for k in time interval i^{th} and $dn_i(k) \in (|B.d|, |f(h(k))|)$ as shown in (1). Furthermore, the parameters β and γ can be obtained form SLA, which measures the importance of routing table and correctness, respectively. $FTLB$ gets out the keys one by one along the order until the overloaded tasks disappear(Line 3–9). $M(k)$ is the function to order keys and get the best migration candidate (Line 5). The selected candidates are inserted into the candidate set $\mathcal{C}$. This step will not stop until no instances are overloaded.

In the second phase, we distribute the loads in $\mathcal{C}$ to our instances. First, we take the same process as $LLFD$ (Line 12–14). The load loading to instance d may cause the overload of d

(Line 15). In such a case, we have to remove some loads to make d un-overloaded. In order to improve the process efficiency, we define *cursor* for each instance to guide the selection of loads on instance d (Line 16–17). Those cursors monitor the maximum workload that could be migrated out of the instance d which shall be less than $cursor[d]$. If instance d is still overloaded (Line 20) in Algorithm 3, $FTLB$ will cancel this insertion of k to instance d (Line 21). And then select the second least loaded instance and redo from Line 14 in Algorithm 3. In this way, $FTLB$ will be able to find a suitable instance for $c(k)$.

After the new workload assignment plan has been generated by Algorithm 3, we do the actual migration action as the procedure shown in Algorithm 4. To generate the migration key set $\mathcal{K}_i$, Algorithm 4 first generate the routing function F_i for workload in time interval i^{th} according to (1) (Line 1). Then, KSM find the different items between F_i and F_{i-1} (Line 2) according to (4) to generate the candidate migration key set $\mathcal{K}_i$. Finally, keys belonging to $\mathcal{K}_i$ will be migrated (Line 3).

Algorithm 4 Pseudo code of key state migration (KSM).

Input: F_{i-1}, $\mathcal{R}_i$ and implicit hash $h_i(k)$
Output: Balance state
1 $F_i \leftarrow \mathcal{R}_i \bigcup h_i$, according to (1);
2 $\mathcal{K}_i \leftarrow F_i \bigoplus F_{i-1}$, according to (4);
3 Migrate states of keys in $\mathcal{K}_i$;

4.2.3 Implementation and cost analysis

Generally, workload adjustment among tasks takes the following steps as follows:

B1 Load Reporting: each task reports current workload to Controller, and Controller checks overall system balance status;
B2 Detailed Information Asking: If unbalanced, Controller requires all instances report detailed information;
B3 Detailed Information Collecting: Instances report their detailed load information to Controller, including key id, located task id, migration cost and workload information;
B4 Migration Plan Generation: Controller makes balancing plan. Keys which will be migrated should be buffered and all $\mathcal{R}_i$ in upstream instances should be updated;
B5 Migrating: System runs migrate action among tasks;
B6 Re-launching: System starts all Paused job again.

According to the balancing framework, we give a detailed cost analysis in the following content.

Preparation phase B1∼ B3 compose the cost for this phase with $B3$ introducing the main cost. For example, assuming each report item to the controller is 8 Byte, when K is lager such as 10^8, the network transmission cost is about 800MB, which is considerable quantity.

Plan construction phase As described in Section 4.2.2, $FTLB$ takes two-step strategy for load balance with the predefined routing table size. Balance migration plan generation $B4$ includes two functions: generating moving candidates (GMC) and pairing load-instance. The computation complexity for GMC is $O(\log_2 \frac{K}{|\mathcal{D}|})$. Moreover, in order to balance the overloaded instances, the overall computation complexity is $O(\log_2 \frac{K}{|\mathcal{D}|} * K * \log_2 N)$. For $K \gg |\mathcal{D}|$, then the computation complexity is $O(Klog_2 K)$.

Migration phase $B4 \sim B6$ generate cost in this phase. If processes are paused when generating migration schedule on upstream instances, it may lead to job ceasing in downstream instances. So a tidy and small routing table is required to shorten migration time and improve processing efficiency.

When unbalance occurs on stream system, it is expected to complete balancing quickly and correctly. A lightweight computing model is required. In this section, though we expect to design algorithms to achieve load balance with small migration cost and limited routing table, our balancing framework shows that either computational complexity or transfer cost is decided by the size of K, huge usually. To reduce the computation complexity, we design new models based on load organization as shown in [10].

5 Evaluation

In this section, we evaluate our proposals by comparing against a handful of baseline approaches. All of these approaches are implemented and run on top of *Apache Storm* [1] under the same task configuration and routing table size. The *Storm* system (in version 0.9.3) is deployed on a 21-nodes HP blade cluster with CentOS 6.5 operating system. Each node is equipped with two Intel Xeon processors (E5335 at 2.00 GHz) containing four cores and 16 GB RAM in total. Each core is exclusively bounded to a worker thread during our experiments.

We use the consistent hashing [21] as our basic hash function. On the other hand, to simulate the computation cost for both simple and complicated tasks, we manipulate the CPU processing time for each key by adding a constant delay to the processing (e.g. one millisecond per tuple) so as to get the maximum computation capacity of each node. We also set the *setMaxSpoutPending(50)* to control the speed of the input tuple flow. We run experiments on both synthetic and real data.

5.1 Experiment settings

5.1.1 Query and dataset

We perform the TopK query on two datasets: a synthetic dataset and a real-world dataset, respectively. In terms of the synthetic one, we use a data generator which can generate skewed dataset and generate a test dataset satisfying Zipf distribution. The overall data skewness is controlled by factor z, and the data distribution in each task is dynamically changed with time, which is configured by the factor f. Such a continuously changing input flow requires the system to balance the workload consistently in order to reduce the unbalance rate down to the bearing threshold $\theta_{\max}$. We also test our performance on a real-world dataset *Social* obtained from social network *Weibo*. The *Social* dataset contains 10 days of Weibo posts, which is about 20 GB and contains roughly 2×10^6 tuples after the text processing. The experiments on *Social* dataset is to analyse the hottest topic. Therefore, the topology of this query consists of three stages: topic extraction, topic count and aggregation. The first operation is mapping operation, while the latter two are aggregation operations.

5.1.2 Baseline comparison

We separate the baseline method into two parts: workload-balancing and fault tolerance. The baseline of workload-balancing method was mentioned in [10] which is called *Mixed*

method, we also use *MinTable* algorithm(MRT) in the meantime. The experiment results in [10] shows that the *MinTable* algorithm tries to generate a migration plan that minimizes the size of the routing table, and makes the migration procedure as simple as possible. Hence, this method performs well in terms of the efficiency, and we aim to compare our efficiency with it. Meanwhile, we also use *Readj* as our baseline method. The *Readj* method pairs the task instance with the key and create workload balance plan. For each pair, this method considers all possible combinations so as to find the best migration plan which reduces the load unbalance. In terms of the fault-tolerance solutions, since we are the first to propose data-level fault-tolerance, we compare our method to the task-level backup method TaskB(Copy Whole Task) regarding the resource utilization rate and efficiency.

5.1.3 Evaluation metrics

We use the following metrics to evaluate the performance. The **Workload Skewness** is calculated by the maximum workload over average workload within one task instance, denoted by $\frac{\max L(d)}{\overline{L}}$. The **Migration Cost** evaluates the percentage of the keys and its states that is migrated during data migration for all tasks. The **Average Generation Time** is the average time spent on making migration plan. The **Recovery Latency** is the average delay of the tuples being recovered. The **Throughput** is the average number of tuples processed during a unit of time. The following section will report the average value of the above metrics based on the complete workflow so as to evaluate the advantages of different methods.

5.2 Experiment results and analysis

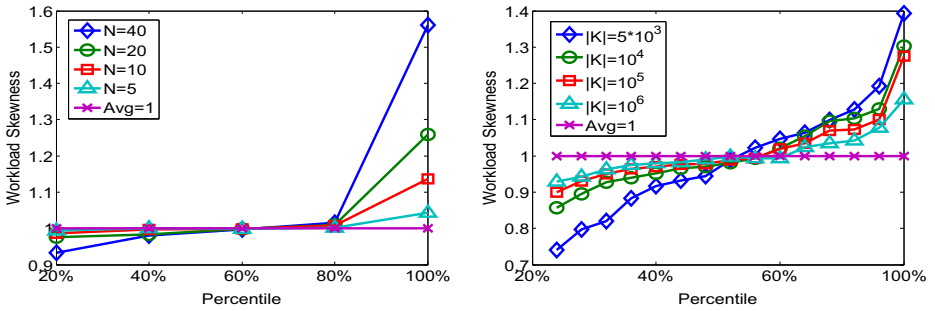
We first show the workload unbalance phenomenon under the data-level fault-tolerance strategy. Then we further evaluate our performance when the balance condition changes or the failure happens.

5.2.1 Data skew

To understand how the replicas affect the workload in the data-level fault-tolerant method, we test the performance by manipulating the workload skewness, which can be achieved by changing the number of tasks and the value range of the keys, shown in Figure 7. In Figure 7, the tasks are sorted according to their workloads, a point on the x axis represents the percentile that a task is located according to its workload in the sorted task list. From the overall workload perspective, Figure 7a also indicates that the task data skewness gains when the number of tasks increases. However, in terms of the task with the largest workload, its skewness becomes lower than Figure 7 regardless the change of the number of tasks or keys. Therefore, the static data distribution strategy can cause the skewness among parallel tasks, while the data-level replication strategy can somehow relieve such issue.

5.2.2 Workload balance performance

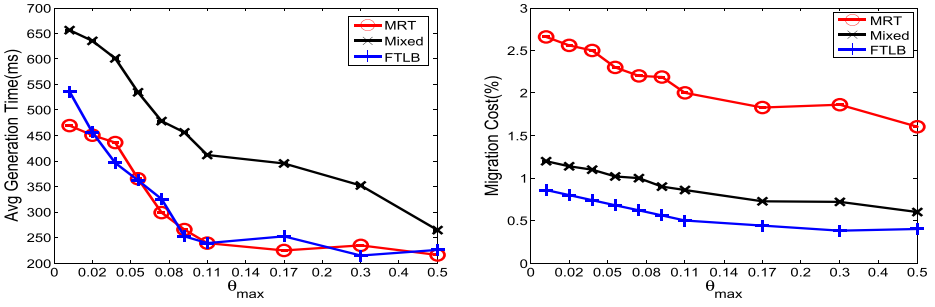
The Figure 8 shows the time spent on making migration plan and the migration cost using different methods when changing the unbalance bearing threshold $\theta_{\max}$. As the aforementioned definition of unbalance bearing threshold, the greater the $\theta_{\max}$ is, the more unbalance the system can afford. In other words, a higher $\theta_{\max}$ makes the system much easier to be balanced, and the migration process runs much faster on a synthetic dataset. Hence, the overall time cost is reduced, shown in Figure 8a. When the $\theta_{\max} < 0.2$, since the *Mixed* algorithm



(a) # of task instances (b) Size of key domain

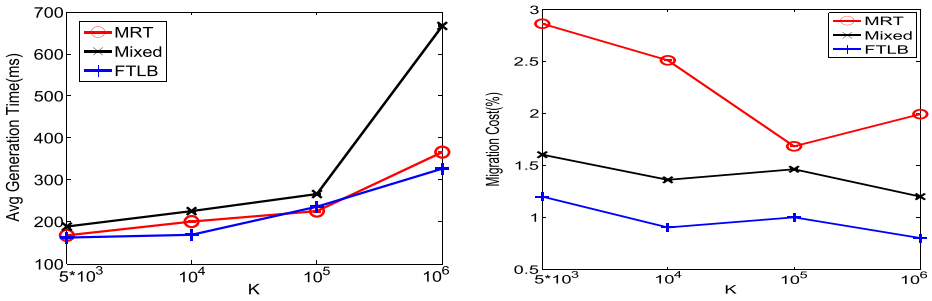
Figure 7 Cumulative distribution of workload skewness under hash-based scheme

has to call the heuristic method iteratively to generate better migration plan, the overall cost is relatively higher. When the $\theta_{\max} \geq 0.2$, the system is much easier to be balanced. Therefore, the effectiveness of *Mixed* is equivalent to the *MRT*(*MinTable* algorithm). However, although the *FTLB* uses the same heuristic method to generate better migration plan, its time spent on making plans is still shorter due to more available keys (keys from backup data) participated the balancing process. The Figure 8b shows the migrated data sizes using different migration planner. The smaller $\theta_{\max}$ usually means the system has to move more data in order to achieve the balance and therefore leads to higher migration cost. The trend in Figure 8b proves such analysis. Since different migration planner focus on different principle, the *MRT* spends higher migration cost than the *Mixed* algorithm when in the same configuration. Meanwhile, since the *FTLB* consider moving backup data instead of moving active one, it can achieve even lower migration cost. Overall, the Figure 8 shows that our method outperforms other baselines in terms of both the time cost on generating migration plan and the cost of data migration. Even when $\theta_{\max} = 0.02$, which is close to absolute balance, our *FTLB* algorithm can still generate the plan within one second. Moreover, since our method utilizes the backup copies, we can find more possible solutions to load balancing problem and the role switching between active and backup data helps reduce the migration cost significantly.



(a) θ_{max} *vs* scheduling efficiency (b) θ_{max} *vs* migration cost

Figure 8 Performance under different $\theta_{\max}$



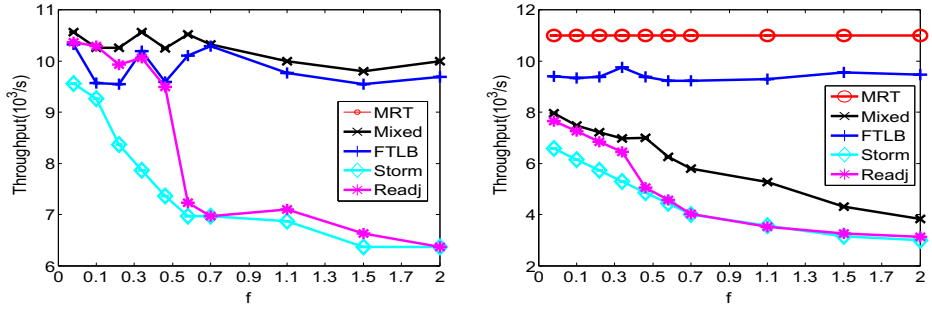
(a) K *vs* scheduling efficiency (b) K *vs* migration cost

Figure 9 Performance under different key numbers

As shown in the above Figure 7b, the number of distinct keys in the dataset affects the skewness of the tasks, the Figure 9 shows the influence on the migration planning time and migration cost under different cardinalities of keys K.

We test the cases that when $K \in [5,000, 1,000,000]$, the data skewness is $z = 0.8$. The Figure 9a shows the change of migration planning cost when changing K. As the number of keys increases, the cost of all methods increases as well. It is because more keys available means the planner will consider moving more keys to achieve a better balancing performance. Among all solutions, the *Mixed* method shows a relative high migration planning cost since it uses a heuristic method to find better migration plan through iteration, shown in Figure 8. Meanwhile, although the *FTLB* method finds the migration plan in a similar way as *Mixed*, its time cost is much closer to *MRT* algorithm, shown in Figure 8a, due to its considerably larger number of viable balancing solutions. In contrary, the Figure 9b shows that with the help of replications, our *FTLB* algorithm achieves an overall lower migration cost under different cardinality of K.

The Figure 10a denotes the throughput of different methods when no task failure is happening. The *Ideal* represents the optimal throughput, which can be achieved when the workload is shuffled randomly. However, a random distribution of tuples does not preserve the semantic meaning of the key-based operations. Hence, although having the optimal throughput, the *Ideal* method does not fit for state-based aggregation operations. Therefore, we regard its throughput as the ideal scenario and compare the other methods with it. The



(a) Throughput without fault task (b) Throughput with fault task

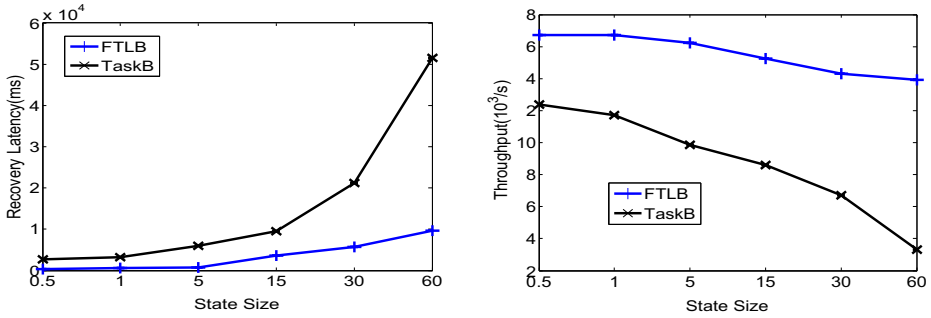
Figure 10 Throughput performance

performance of those methods, except *FTLB*, when changing the distribution fluctuation frequency f has the similar trend of performance degradation. Note that the throughput of *FTLB* is slightly lower than the *Mixed* method in Figure 10a as the extra cost on maintaining the state of replications increases the latency of processing the tuples, and thus affects the overall throughput. However, since the *Mixed* method sacrifices the fault-tolerance feature, its throughput is overwhelmed by our method once the failure happens, which will be discussed in the following subsection.

5.2.3 Fault-tolerance performance

In the Figure 10b, the throughput of each method is compared when the task failure happens. We configure the $message.timeout.secs$ and $acker$ settings in the Storm configuration file to ensure every record is processed, and we obstruct the output of tuples in one of the parallel tasks to simulate task failure scenario. If the obstructed tuple has replica, we stop the resend of the replicated tuple by changing the value of its $acker$. The *Ideal* in Figure 10b represents the optimal throughput when no task failure happens, and the workload is absolute balanced. Compared to the case in Figure 10a where no task failure happens, the performance of both *Mixed*, *Readj* and *Storm* decreases significantly when a task fails. Since the fault-tolerant mechanisms of those methods purely rely on the in-built fault tolerance of Storm, which is achieved by redistributing the time-out tuples. The Storm system keeps monitoring the data output port and decides whether to resend a tuple or not according to the preset maximum time limit. If the system does not receive a "processed successfully" message from a processing tuple within the given time, it will mark this tuple as lost and resend it. Since such way of fault tolerance is guaranteed by resending the failed data, it can significantly increase the latency. Moreover, the severity of the latency is determined by the topological position of the failed task and the active duration of the failed data. For instance, if the failed task located at the last step of the topology, the recovery usually requires more recalculation. Similarly, if the failed data are correlated to many other data, the recovery also incurs a huge cost as all the relevant data have to be recalculated. Overall, our proposed method *FTLB* achieves the best throughput in Figure 10b, and it did not lose much throughput when varying workload fluctuation frequency despite a few errors sent, shown in Figure 10a.

The Figure 11 shows the recovery latency and throughput of various methods when task failure happens. The recovery method in *TaskB* focus on task-level fault tolerance. When the failure happens, it activates the backup task to continue the data process. The x-axis in the



(a) Recovery latency in different state size (b) Throughput in different state size

Figure 11 Fault-tolerance performance

figure represents the size of the data state in the operation buffer when the system is running, which is controlled by varying the size of the data buffer time windows from 0.5 min to 1 hr. The Figure 11a compares the recovery latency between the data-level fault-tolerant method *FTLB* and the task-level method. *TaskB*.

As shown in the figure, the *TaskB* has an overall higher latency than *FTLB* since the *FTLB* sends the backup data to all the tasks that contain failed data simultaneously so as to do the recover locally and continue the data process. Moreover, compared with task-level backup method *TaskB*, the *FTLB* recovers less data. The Figure 11b shows a similar trend as in Figure 11a in terms of the average throughput under varies buffer size when dealing with fixed size dataset. In general, the experiments indicate that the data-level fault-tolerant methods outperform the task-level solution in throughput.

6 Related work

Since the increasing of stream data, processing task can easily run out of computation resources. Distributed stream processing over the shared nothing architecture has been a scalable processing candidate. However, if we do not have the global task processing cost information or data skew information, workload distribution will meet with imbalance. It will bring down the whole processing ability, e.g. low throughput or high latency. Dynamic load balancing technology is essential for promising system usability. Existing work usually splits high-performance and high-availability in real-time processing systems into two parts:

Workoad balance Generally, there are two kinds of balancing mechanisms: *operator-level* and *data-level*. The operator-level balancing migrates the whole operator from one task to another, with the assumption that each operator can be assigned to one machine. In contrast, the data-level balancing groups the data into migration units and processes them separately during the runtime. **1)**In terms of the operator-level balancing, the basic System S [36] performs the scheduling during the job submission. It then migrates the jobs or sub-jobs to machines with less load during runtime according to the task workload, operators workload and the priority of the applications. Khandekar et al. [23] combines multiple operators to reduce communication. In order to improve the balance property of storm, [3] presents a more flexible method which includes online and offline manner to minimize the network traffic. **2)**Regarding the data-level balancing, [24] and [9] both primarily focus on designing a distributed scalable stream join processing model. Elseidy et al. [9] designs an adaptive balancing strategy on the square matrix, each side of which corresponds to the partitions of one relation. To avoid high usage of memory and network, [24] proposes a join-biclique model that takes a large cluster as a complete bipartite graph for joining big data streams. For key grouping, [25, 26] introduces a new stream partitioning scheme for balanced key grouping, namely Partial Key Grouping(PKG). This solution applies the power of *two-choices-approach* to provide better load balancing. Gedik [12] introduces its partitioning function which consists of a basic consistent hash function and an explicit hash to solve the stateful load balance problem. Katsipoulakis et al. [22] takes both tuple imbalance and aggregation cost into account to partition incoming stream. Rupprecht et al. [29] lazy partitioning to deal with transient network skew among clusters adaptively to avoid the straggling workers prolonging join completion time.

Fault tolerance In DSPEs, traditional fault-tolerance methods, including active backup, passive backup, or upstream backup, usually apply to all operations in the application in order for a low latency recovery performance. LAAR [5] provides a-priori guarantees about the achievable levels of fault-tolerance, expressed by an internal completeness metric that captures the maximum amount of information that can be lost in case of failures. Su and Zhou [32, 33] propose a new fault-tolerance framework, which is Passive and Partially Active (PPA). In a PPA scheme, the passive approach is applied to all tasks while only a selected set of tasks will be actively replicated. The number of actively replicated tasks depends on the capacity of available resources. If tasks without active replicas fail, tentative outputs will be generated before the completion of the recovery process. Noghabi et al. [27] uses the *Host Affinity* mechanism to reduce the overhead of rebuilding state at a restarted task. This mechanism can reduce the restart time to a constant value, rather than linearly growing with the state size. Salama et al. [30] selects a subset of intermediates to be materialized such that the total query runtime is minimized under mid-query failures. FTOpt [34] is a cost-based fault-tolerance optimizer. It automatically selects the best strategy for each operator in a query plan in a manner that minimizes the expected processing time with failures for the entire query. Similar to the upstream backup, the TimeStream [28] uses a recovery mechanism that tracks which upstream data each operator depends on and replays it serially through a new copy of the operator. In contrast, D-Streams [39] use stateless transformations and explicitly put state in data structures(RDDs) which forgo conventional streaming wisdom by batching data into small timesteps.

7 Conclusion

In this paper, we focused on solving the fault tolerance and workload balancing issues in distributed stream processing system. Different from the previous works that solved these two problems separately, we aimed to generate a solution that achieves both goals in one shot. Despite the inherent characteristics of data streams, such as persistence, dynamic, and unpredictability, we proposed a new dynamic workload distribution mechanism for intra-operator load balancing and fault tolerance which achieved a lower resource consumption compared to the previous methods. To achieve the fault tolerance and make sure the result correctness, we defined synchronization protocols and classified the current popular operations into different categories according to their time dependency. Furthermore, we proposed a rebalanced approach that takes into account the fault-tolerant mechanism. Through extensive experimental verification, it is proved that our algorithm helps the system achieve efficient load balancing and fault tolerance with low computation resource requirement.

Acknowledgments This work is partially supported by National Science Foundation of China under grant (No. 61802273, 61572194, 61772356, 61572335, and 61836007), Postdoctoral Research Foundation of China (2017M621813), Postdoctoral Science Foundation of Jiangsu Province (2018K029C), Natural science fund for colleges and universities in Jiangsu Province (18KJB520044), and Suzhou Science and Technology Development Program(SYG201803). This work is also supported by the Open Program of Neusoft Corporation(SKLSAOP1801) and Blockheaders Co. Ltd.

References

1. Apache Storm. http://storm.apache.org/
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDBJ **12**(2), 120–139 (2003)
3. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: DEBS, pp. 207–218 (2013)
4. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst. (TODS) **33**(1), 3 (2008)
5. Bellavista, P., Corradi, A., Kotoulas, S., Reale, A.: Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In: EDBT, pp. 85–96 (2014)
6. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proceedings of the Integrating Scale ACM SIGMOD International Conference on Management of Data, p. 2013. ACM (2013)
7. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. In: CIDR, vol. 3, pp. 257–268 (2003)
8. Coffman Jr, E.G., Garey, M.R., Johnson, D.S.: Approximation algorithms for bin-packing¡^{a} an updated survey. In: Algorithm Design for Computer System Design, pp. 49–106. Springer (1984)
9. Elseidy, M., Elguindy, A., Vitorovic, A., Koch, C.: Scalable and adaptive online joins. VLDB **7**(6), 441–452 (2014)
10. Fang, J., Zhang, R., Fu, T.Z., Zhang, Z., Zhou, A., Zhu, J.: Parallel stream processing against workload skewness and variance. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 15–26. ACM (2017)
11. Fu, T.Z.J., Ding, J., Ma, R.T.B., Winslett, M., Yang, Y., Zhang, Z.: Drs: dynamic resource scheduling for real-time analytics over fast streams. In: ICDCS, pp. 411–420. IEEE, Columbus (2015)
12. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. VLDBJ **23**(4), 517–539 (2014)
13. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In: 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 716–723. IEEE (2011)
14. Heath, T., Martin, R.P., Nguyen, T.D.: Improving cluster availability using workstation validation. In: ACM SIGMETRICS Performance Evaluation Review, vol. 30, pp. 217–227. ACM (2002)
15. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C.: An adaptive replication scheme for elastic data stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, pp. 150–161. ACM (2015)
16. Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., Zdonik, S.: High-availability algorithms for distributed stream processing. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pp. 779–790. IEEE (2005)
17. Hwang, J.-H., Cetintemel, U., Zdonik, S.: Fast and highly-available stream processing over wide area networks. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, pp. 804–813. IEEE (2008)
18. Hwang, J.-H., Xing, Y., Cetintemel, U., Zdonik, S.: A cooperative, self-configuring high-availability solution for stream processing. In: Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pp. 176–185. IEEE (2007)
19. Jacques-Silva, G., Gedik, B., Andrade, H., Wu, K.-L., Iyer, R.K.: Fault injection-based assessment of partial fault tolerance in stream processing applications. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, pp. 231–242. ACM (2011)
20. Ji, Y., Nica, A., Jerzak, Z., Hackenbroich, G., Fetzer, C.: Quality-driven disorder handling for concurrent windowed stream queries with shared operators. In: Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, pp. 25–36. ACM (2016)
21. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: STOC, pp. 654–663 (1997)
22. Katsipoulakis, N.R., Labrinidis, A., Chrysanthis, P.K.: A holistic view of stream partitioning costs. VLDB **10**(11), 1286–1297 (2017)
23. Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., Gedik, B.: Cola: Optimizing stream processing applications via graph partitioning. In: Middleware, pp. 308–327 (2009)
24. Lin, Q., Ooi, B.C., Wang, Z., Yu, C.: Scalable distributed stream join processing. In: SIGMOD, pp. 811–825 (2015)

25. Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. ICDE (2015)
26. Nasir, M.A.U., Serafini, M., et al.: When two choices are not enough: Balancing at scale in distributed stream processing. In: ICDE (2016)
27. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: Stateful scalable stream processing at linkedin. VLDB **10**(12), 1634–1645 (2017)
28. Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., Zhang, Z.: Timestream: Reliable stream computation in the cloud. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 1–14. ACM (2013)
29. Rupprecht, L., Culhane, W., Pietzuch, P.: Squirreljoin: Network-aware distributed join processing with lazy partitioning. Proceedings of the VLDB Endowment **10**(11), 1250–1261 (2017)
30. Salama, A., Binnig, C., Kraska, T., Zamanian, E.: Cost-based fault-tolerance for parallel data processing. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 285–297. ACM (2015)
31. Schroeder, B., Gibson, G.: A large-scale study of failures in high-performance computing systems. IEEE Trans. on Dependable and Secure Comput. **7**(4), 337–350 (2010)
32. Su, L., Zhou, Y.: Tolerating correlated failures in massively parallel stream processing engines. In: ICDE, pp. 517–528 (2016)
33. Su, L., Zhou, Y.: Passive and partially active fault tolerance for massively parallel stream processing engines. TKDE (2017)
34. Upadhyaya, P., Kwon, Y., Balazinska, M.: A latency and fault-tolerance optimizer for online parallel query plans. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 241–252. ACM (2011)
35. Vishwanath, K.V., Nagappan, N.: Characterizing cloud computing hardware reliability. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 193–204. ACM (2010)
36. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer, L.: Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In: Middleware, pp. 306–325 (2008)
37. Xing, Y., Hwang, J., Cetintemel, U., Zdonik, S.: Providing resiliency to load variations in distributed stream processing. In: VLDB, pp. 775–786 (2006)
38. Xing, Y., Zdonik, S., Hwang, J.: Dynamic load distribution in the borealis stream processor. In: ICDE, pp. 791–802 (2005)
39. Zaharia, M., Das, T., Li, H., Hunt, T.: 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)