

# Parallel Stream Processing Against Workload Skewness and Variance

Junhua Fang<sup>†</sup> Rong Zhang<sup>†</sup> Tom Z.J.Fu<sup>‡</sup> Zhenjie Zhang<sup>‡</sup> Aoying Zhou<sup>†</sup> Junhua Zhu<sup>◇</sup>

<sup>†</sup>Institute for Data Science and Engineering, Software Engineering Institute,  
East China Normal University, Shanghai, China  
{jh.fang,rzhang,ayzhou}@sei.ecnu.edu.cn

<sup>‡</sup>Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd.  
{tom.fu,zhenjie}@adsc.com.sg

<sup>◇</sup>Huawei Technologies Co. Ltd. {junhua.zhu}@outlook.com

**Abstract**—Key-based workload partitioning is a common strategy used in parallel stream processing engines, enabling effective key-value tuple distribution over worker threads in a logical operator. While randomized hashing on the keys is capable of balancing the workload for key-based partitioning when the keys generally follow a static distribution, it is likely to generate poor balancing performance when workload variance occurs on the incoming data stream. This paper presents a new key-based workload partitioning framework, with practical algorithms to support dynamic workload assignment for stateful operators. The framework combines hash-based and explicit key-based routing strategies for workload distribution, which specifies the destination worker threads for a handful of keys and assigns the other keys with the hashing function. When short-term distribution fluctuations occur to the incoming data stream, the system adaptively updates the routing table containing the chosen keys, in order to rebalance the workload with minimal migration overhead within the stateful operator. We formulate the rebalance operation as an optimization problem, with multiple objectives on minimizing state migration costs, controlling the size of the routing table and breaking workload imbalance among worker threads. Despite of the NP-hardness nature behind the optimization formulation, we carefully investigate and justify the heuristics behind key (re)routing and state migration, to facilitate fast response to workload variance with ignorable cost to the normal processing in the distributed system. Empirical studies on synthetic data and real-world stream applications validate the usefulness of our proposals and prove the huge advantage of our approaches over state-of-the-art solutions in the literature.

## I. INTRODUCTION

Workload skewness and variance are common phenomena in distributed stream processing engines. When massive stream data flood into a distributed system for processing and analyzing, even slight distribution change on the incoming data stream may significantly affect the system performance. Existing optimization techniques for stream processing engines are designed to exploit the distributed processor, memory and bandwidth resources based on the computation workload, but potentially generate suboptimal performance when the evolving workload deviates from expectation. Unfortunately, workload evolution is constantly happening in real application scenarios (e.g., surveillance video analysis [6] and online advertising monitoring [18]). It raises new challenges to distributed system on solutions to handle the dynamics of data

stream while maintaining high resource utilization rate at any time.

In distributed stream processing system, abstract operators are connected in form of a directed graph to support complex processing logics over the data stream. Traditional load balancing approaches in distributed stream processing engines attempt to balance the workload of the system, by evenly assigning a variety of heterogeneous tasks to distributed nodes [3], [4], [16], [31], [32]. Such strategies may not perform as expected in distributed stream processing systems, because of the lack of balance on the homogeneous tasks within the same abstract operator. In Fig. 1, we present an example to illustrate the potential problem with such strategies. In the example, there are three logic operators in the pipeline, denoted by rectangles. There are three concrete task instances running in *operator 2*, denoted by circles. The number of incoming tuples to the first task instance is two times of that to the second and third task instances, due to the distribution skewness on the tuples. Even if the system allocates the tasks in a perfect way to balance the workload when allocating task instances to computation nodes, the processing efficiency may not be optimal. Because of the higher processing latency in the first task instance of *operator 2*, *operator 1* is forced to slow down its processing speed under backpushing effect, and *operator 3* may be suspended to wait for the complete intermediate results from *operator 2*. This example shows that load balancing between task instances within individual logical operators is more crucial to distributed stream processing engines, to improve the system stability and guarantee the processing performance.

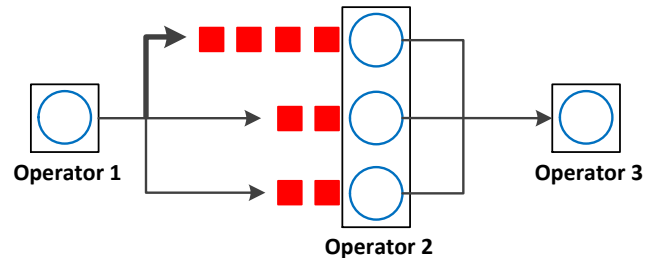


Fig. 1. The potential problem of workload imbalance within operators in real distributed stream processing engine.

There are two types of workload variance in distributed stream processing engines, namely *long-term* workload shift and *short-term* workload fluctuation. Long-term workload shifts usually involve distribution changes on incoming tuples driven by the change in physical world (e.g., regular burst of tweets after lunch time), while workload fluctuations are usually short-term and random in nature. Long-term workload shifts can only be solved by applying heavyweight resource scheduling, e.g., [10], which reallocates the computation resource based on the necessity. Computation infrastructure of the distributed system may request more (less resp.) resource, by adding (returning resp.) virtual machines, or completely reshuffling the resource between logical operators according to the demands of operators. Such operations on the infrastructure level are inappropriate for short-term workload fluctuations, usually too expensive and render suboptimal performance when the fluctuation is over. It is thus more desirable to adopt lightweight protocols within system, to smoothly redistribute the workload between task instances, minimize the impact on the normal processing, and achieve the objective of load balancing within every logical operator. This paper focuses on such a dynamic workload assignment mechanism for individual logical operators in a complex data stream processing logic, especially against short-term workload fluctuations. Note that existing solutions to long-term workload shifts are mostly orthogonal to the mechanisms for short-term workload fluctuations, both of which can be invoked by the system optionally based on the workload characteristics.

The methods which focus on designing load balance algorithms for stream system can be divided into two categories: **split-key-based** and **non-split-key-based**. Split-key-based method splits data of the same key and distributes them to the parallel processing instances. The representative algorithm is implemented in PKG [21]. However, this approach distorts the semantics of key-based operations, resulting in additional processing on some operators. In PKG, for example, it splits data of the same key into multiple subsets and distributes them selectively to different instances to avoid load imbalance. However, this approach leads to some negative impact on system performance. As shown in Fig. 2(a), aggregations must contain some partial result operators and an additional merge operator if implemented on a split-key-based architecture. In Fig. 2(b), it shows the join operation on this architecture. The data of key  $k_1$  from  $R$  stream are split into two parts and assigned to instances  $d_1$  and  $d_2$ . To ensure the correctness of join operation, as the routing in work [20], data of  $k_1$  from stream  $S$  should be broadcasted to both instances  $d_1$  and  $d_2$ .

Non-split-key-based methods can guarantee the semantics of Key-based operation, but it weakens system balance capability severely. The representative algorithm is implemented in *Readj* [11]. It treats the data of a key as a complete granularity and does not split it during load adjustment. However, when the number of keys is huge, *Readj* is of high time complexity for it considers all possible swaps by pairing tasks and keys to find the best key movement to alleviate the workload imbalance. Furthermore, *Readj* just considers to adjust the big load keys to make workload balance, which will incur expensive migration cost for stateful operator, such as join.

Then it is a urgent to design a new method which can not only guarantee the key-based semantics, but also balance

system workload quick and efficient.

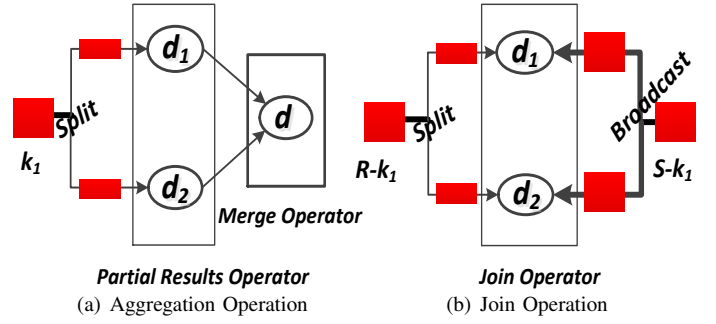


Fig. 2. Sketch for operations that using split-key-based to make workload balance.

Our proposal in this paper is based on a mixed strategy of key-based workload partitioning, which explicitly specifies the destination worker threads for a handful of keys and assigns all other keys with the randomized hashing function. This scheme achieves high *flexibility* by easily redirecting the keys to new worker threads with simple editing on the routing table. It is also highly *efficient* when the system sets the maximal size of the routing table, thus controlling the memory overhead and calculation cost with the routing table. Workload redistribution with the scheme is *scalable* and *effective*, by allowing the system to respond promptly to the short-term workload fluctuation even when there are a large number of keys present in the incoming data stream. To fully unleash the power of the scheme, it is important to design a monitoring and controlling mechanism on top of system, making optimal decisions on routing table update to achieve intra-operator workload balancing. Recent research work in [11], although employing similar workload distribution strategy, only considers migration of *hot* keys with high frequencies, limiting the optimizations within much smaller configuration space. We break the limit in this paper with a new solution for distributed systems to explore possible optimizations with all candidate keys for the routing table, thus maximizing the resource utilization with ignorable additional cost. Specifically, the technical contributions of this paper include:

- We design a general strategy to generate the partition function for data redistribution under different stream dynamic changes at runtime, which achieves scalability, effectiveness and efficiency by a single shot.
- We propose a lightweight computation model to support rapid migration plan generation, which incurs minimal data transmission overhead and processing latency.
- We present a detailed theoretical analysis for proposed migration algorithms, and prove its usability and correctness.
- We implement our algorithms on Storm and give extensive experimental evaluations to our proposed techniques by comparing with existing work using abundant datasets. We explain the results in detail.

The remainder of this paper is organized as follows. Section II introduces the overview and preliminaries of our problem.

Section III presents our balancing algorithms to support our mixed workload distribution scheme. Section IV proposes the optimization techniques used in the implementation of our proposal. Section V presents empirical evaluations of our proposal. Section VI reviews a wide spectrum of related studies on stream processing, workload balancing and distributed systems. Section VII finally concludes the paper and addresses future research directions.

## II. PRELIMINARIES

A distributed stream processing engine (DSPE) deploys abstract stream processing logics over interconnected computation nodes for continuous stream processing. The abstract stream processing logic is usually described by a directed graphical model (e.g., Storm [26], Heron [17] and Spark Streaming [34]), with a vertex in the graph denoting a computation operator and an edge denoting a stream from one operator to another. Each data stream consists of key-value pairs, known as *tuples*, transmitted by network connection between computation nodes. The computation logic with an operator is a mapping function with an input tuple from upstream operator to a group of output tuples for downstream operators.

To maximize the throughput of stream processing and improve the utilization rate of the computation resource, the workload of a logical operator is commonly partitioned and concurrently processed by a number of threads, known as *tasks*. The upstream operator is aware of the concrete tasks and sends the output tuples to the tasks based on a global partitioning strategy. All concrete tasks within an operator process the incoming tuples independently. Key-based workload partitioning is now commonly adopted in distributed stream processing engines, such that tuples with the same key are guaranteed to be received by the same concrete task for processing. An operator is called *stateful operator*, if there is a memory space used to keep intermediate results, called *states*, of the keys based on the latest tuples. Basically, a *state* is associated with an active key in the corresponding task in a stateful operator, which is used to maintain necessary information for computation. The state, for example, can be used to record the counts of the words or recent tuples in the sliding window. Because of the tight binding between key and state, when a key is reassigned to another task instance, its state must be migrated as well, in order to ensure the correctness of computation outcomes.

The workload partitioning among concrete tasks is the model as a mapping from key domain to running tasks in the successor operator. A straightforward solution to workload partitioning is the employment of hashing function (e.g., by consistent hashing), which chooses a task for a specific key in a random manner. The computational cost of task selection for a tuple is thus constant. As discussed in previous section, despite of the huge advantages of hashing on memory consumption and computation cost, such scheme may not handle well with workload variance and key skewness. Another option of workload distribution 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 an approach is more flexible on dynamic

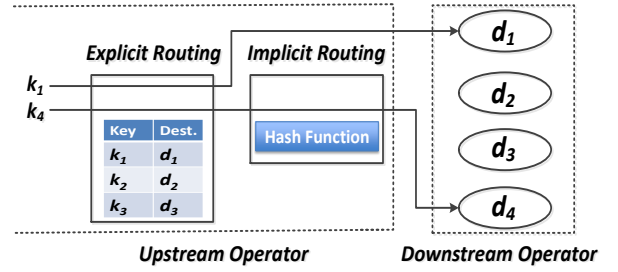


Fig. 3. The scheme of mixed routing with a small routing table and a hash function.

workload repartitioning, the operational cost on both memory and computation is too high to afford in practice.

In this paper, we develop a new workload partitioning framework based on a mixed routing strategy, expecting to balance the hash-based randomized strategy and key-based routing strategy. In Fig. 3, we present an example of the strategy with one data stream between two operators. A routing table is maintained in the system, but contains routing rules for a handful of keys only. When a new output tuple is 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 concrete task instance specified by the entry, otherwise a hashing function is applied on the key to deterministically generate the target task id for the tuple. By appropriately controlling the routing table with a maximal size constraint, both the memory and computation cost of the scheme are acceptable, while the flexibility and effectiveness are achieved by updating the routing table in response to the evolving distribution of the keys.

As illustrated in the previous section, workload balancing between tasks from the same operator is crucial and it is the major problem we aim to tackle with. With the mixed routing strategy, we can solve the problem by focusing on the construction and update of the routing table with the constrained size, without considering the global structure of processing topology and workload. Therefore, our discussion in the following sections is focuses on one single operator and its routing table. Note that our approach is obviously applicable to complex stream processing logics, as evaluated in the experimental section.

### A. Data and Workload Models

In our model, the time domain is discretized into intervals with integer timestamps, i.e.,  $(T_1, T_2, \dots, T_i, \dots)$ . At the  $i$ -th interval, given a pair of upstream operator  $U$  and downstream operator  $D$ , we use  $\mathcal{U}$  and  $\mathcal{D}$  to denote the set of task instances within upstream operator  $U$  and downstream operator  $D$ , respectively. We also use  $N_U = |\mathcal{U}|$  and  $N_D = |\mathcal{D}|$  to denote the numbers of task instances in  $U$  and  $D$ , respectively. A tuple is tuple  $\tau = (k, v)$ , in which  $k$  is the key of the tuple from key domain  $\mathcal{K}$  and  $v$  is the value carried by the tuple. We assume  $N_U$  and  $N_D$  are predefined without immediate change. The discussion on dynamic resource rescheduling, i.e., changing  $N_U$  and  $N_D$ , is out of the scope of this paper, since it involves orthogonal optimization techniques on global resource scheduling (e.g., [10]).

A key-based workload partitioning 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 hashing function  $h : \mathcal{K} \rightarrow \mathcal{D}$  is available to the system for general key assignment. A typical implementation of hash function is consistent hashing, which is believed to be a suitable option for balancing keys among instances. However, it does not consider so much about key granularities which is the number of data of the same key. Hence, the balanced state of parallel processing has to be obtained by moving keys among instances even with the consistent hash and we use routing table to record the mappings from keys to processing destinations which are not the basic hash destinations. A routing table  $A$  of size  $N_A$  contains a group of pairs from  $\mathcal{K} \times \mathcal{D}$ , specifying the destination task instances for keys existing in  $A$ . The mixed routing strategy shown in Fig. 3 is thus modelled by the following equation:

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

Therefore, workload redistribution is enabled by editing the routing table  $A$  with an assignment function  $F(\cdot)$ . In the following, we provide formal analysis on the general properties of the assignment function  $F(\cdot)$ .

**Computation Cost:** We use  $g_i(k)$  to denote the frequency of tuples with key  $k$  in time interval  $T_i$ , and define the computation cost  $c_i(k)$  by the amount of CPU resource necessary for all these tuples with key  $k$  in time interval  $T_i$ . Generally speaking,  $c_i(k)$  increases with the growth of  $g_i(k)$ . Unless specified, we do not make any assumption on the correlation between  $g_i(k)$  and  $c_i(k)$ , both of which are measured in the distributed system and recorded as statistics, in order to support decision making on the update of  $F(\cdot)$ . The total workload with a task instance  $d$  in downstream operator  $D$  within time interval  $T_i$  is calculated by  $L_i(d, F) = \sum_{\{k | F(k)=d, k \in \mathcal{K}\}} c_i(k)$ .

**Load Balance:** Load balance among task instances of the downstream operator  $D$  is the essential target of our proposal in this paper. Specifically, we define the balance indicator  $\theta_i(d, F)$  for task instance  $d$  under assignment function  $F$  during time interval  $T_i$  as  $\theta_i(d, F) = \frac{|L_i(d, F) - \bar{L}_i|}{\bar{L}_i}$ , 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  $\mathcal{D}$ . As it is unlikely to achieve absolute load balancing with  $\theta_i(d, F) = 0$  for every task instance  $d$ , an upper bound  $\theta_{\max}$  is usually specified by the system administrator, such that the workload of task instance  $d$  is approximately balanced if  $\theta_i(d, F) \leq \theta_{\max}$ .

**Memory Cost:** For stateful operators, the system is supposed to maintain historical information, e.g., statistics with the keys, for processing and analysing on newly arriving tuples. We assume that each operator maintains states independently on individual time interval  $T_i$  and only the last  $w$  time intervals are needed by any task instance. It means that the task instance erases the state from time interval  $T_{i-w}$  after finishing the computation on all tuples in time interval  $T_i$ . This model is general enough to cover almost all continuous stream

TABLE I. TABLE OF NOTATIONS

Notations	Description
$T_i$	The $i$ -th time interval
$U$	Upstream operator
$D$	Downstream operator
$N_U, N_D$	Numbers of task instances in $U$ and $D$
$\mathcal{D}$	Instances set of downstream operator
$(k, v)$	Key-value pair on the data stream
$k$	Key of the tuple
$v$	Value of the tuple
$d$	Task instance in downstream operator
$A$	The routing table available to $U$
$N_A$	Number of entries in $A$
$c_i(k)$	Computation cost of all tuples with key $k$ in $T_i$
$g_i(k)$	Frequency of key $k$ in time interval $T_i$
$L_i(d, F)$	Total workload of task instance $d$ under assignment function $F$
$\bar{L}_i$	Average load of all instances in $U$ in time interval $T_i$
$\theta_i(d, F)$	Load balance factor of task instance $d$
$\theta_{\max}$	Upper bound of imbalance tolerance
$S_i(k, w)$	Memory cost of key $k$ with $w$ time intervals at $T_i$
$\Delta(F, F')$	Keys with different destination under $F$ and $F'$
$M_i(w, F, F')$	Total migration cost by replacing $F$ with $F'$ at time interval $T_i$

processing and analytical jobs (e.g., stream data mining over sliding window). The memory consumption for tuples with key  $k$  in  $T_i$  is thus measured as  $s_i(k)$ , and the total memory consumption for key  $k$  is the summation over last  $w$  intervals on the time domain, as  $S_i(k, w) = \sum_{j=i-w+1}^i s_j(k)$ .

**Migration Cost:** Upon the revision on assignment function  $F$ , certain key  $k$  may be moved from one task instance to another. The states associated with key  $k$  must be moved accordingly to ensure the correctness of processing on following tuples with key  $k$ . The migration cost is thus modelled as the total size of states under migration. By replacing function  $F$  with another function  $F'$  at time interval  $T_i$ , we use  $\Delta(F, F') = \{k \mid F(k) \neq F'(k), k \in \mathcal{K}\}$ . The key state migration includes all the historical states within the given window  $w$ . Thus, the total migration cost, denoted by  $M_i(w, F, F')$ , can be defined as:

$$M_i(w, F, F') = \sum_{k \in \Delta(F, F')} S_i(k, w). \quad (2)$$

All notations used in the rest of the paper are summarized in Tab. I.

## B. Problem Formulation

Based on the model of data and workload, we now define our dynamic workload distribution problem, with the objectives on (i) load balance among all the downstream instances; (ii) controllable size on the routing table; and (iii) minimization on state migration cost. These goals are achieved by controlling the routing table in the assignment function, under appropriate constraints for performance guarantee. Specifically, to construct a new assignment function  $F'$  as a replacement for  $F$  in

time interval  $T_i$ , we formulate it as an optimization problem, as below:

$$\begin{aligned} \min_{F'(\cdot)} \quad & M_i(w, F, F') \\ \text{s.t.} \quad & \theta(d, F') \leq \theta_{\max}, \forall d \in \mathcal{D}, \\ & N_A \leq A_{\max}, \end{aligned} \quad (3)$$

in which  $F$  is the old assignment function and  $F'$  is the variable for optimization. The target of the program above is to minimize migration cost, while meeting the constraints on load balance factor and routing table size with user-specified balance bounds  $\theta_{\max}$  and  $A_{\max}$  which is the maximum constrained size of  $A$ .

It is worthwhile to emphasize that the new assignment function is constructed at the beginning of a new time interval  $T_i$ . The optimization is thus purely based on the statistical information from previous time interval  $T_{i-1}$ . The metrics defined in previous subsection are estimated with frequencies  $\{g_{i-1}(k)\}$  over the keys, the computation costs  $\{c_{i-1}(k)\}$  and the memory consumption  $S_{i-1}(k, w)$ .

The problem of initializing the keys in  $\mathcal{K}$ , with the task instance set  $\mathcal{D}$  and load balance constraint  $\theta_{\max}$ , is a combinatorial NP-hard problem, as it can be reduced to Bin-packing problem [15]. Even Worse, our optimization problem also puts constraints on the maximal table size and migration cost. Specifically, even if the parallel instances achieve the balance state, but the routing table size exceeds the predefined space limits as shown in Equ. 3, this balanced state should not conform to the requirement. Therefore, in the following section, we discuss a number of heuristics with careful analysis on their usefulness.

### III. ALGORITHMS

In this section, we introduce algorithms to solve the optimization problem raised in previous section, targeting to construct a new assignment function  $F'$  by updating the routing table  $A$ . We simply assume that all necessary statistics are available in the system for algorithms to use. The implementation details, including measurement collection, are discussed in the following section.

Since the optimization problem is clearly NP-hard, there is no polynomial algorithm to find global optimum, unless  $P=NP$ . In the rest of the section, we firstly describe a general workflow for a variety of heuristics, such that all algorithms based on these heuristics follow the same operation pattern. We then discuss a number of heuristics with objectives on routing table minimization and migration minimization. A mixed algorithm is introduced to combine the two heuristics in order to accomplish the constraints in the optimization formulation with a single shot.

Generally speaking, the system follows the steps below when constructing a new assignment function  $F'$ .

**Phase I (Cleaning):** It attempts to *clean* the routing table  $A$  by removing certain entries in the table. This is equivalent to moving the keys in the entries back to the original task instance assignment, decided by the hash function. Different algorithms may adopt different cleaning strategies to shrink the existing routing table in  $F$ . Note that such a temporary removal

does not physically migrate the corresponding keys, but just generates an intermediate result table for further processing.

**Phase II (Preparing):** It identifies candidate keys for migration from overloaded task instances, i.e.,  $\{d | L(d) > L_{\max}\}$ , where  $L_{\max} = (1 + \theta_{\max})\bar{L}$ . Different selection criteria, such as keys with highest computation cost first, and largest computation cost per unit memory consumption first (concerning about migration cost), and etc, can be applied by the algorithm to select keys and disassociate their assignments from the corresponding task instances. These disassociated keys will be temporarily put into a candidate key set (denoted by  $\mathcal{C}$ ) for processing in the third step of the workflow.

**Phase III (Assigning):** It reshuffles the keys in the candidate set by manipulating the routing table, in order to balance the workloads. In particular, all algorithms proposed in this paper invoke the Least-Load Fit Decreasing (LLFD) subroutine, which will be described shortly, in this phase.

---

#### algorithm 1 Least-Load Fit Decreasing Algorithm

---

**input:** key candidate  $\mathcal{C}$ , task instances in  $\mathcal{D}$ , imbalance tolerance factor  $\theta_{\max}$ , key selection criteria  $\psi$

**output:**  $A'$

```

1: foreach  $d$  in  $\mathcal{D}$  do
2:   Initialize estimation  $\hat{L}(d) = L_{i-1}(d)$ 
3: foreach  $k$  in  $\mathcal{C}$  in descending order of  $c_{i-1}(k)$  do
4:   foreach  $d$  in  $\mathcal{D}$  in ascending order of  $L_{i-1}(d)$  do
5:     if  $\text{Adjust}(k, d, \mathcal{C}, \theta_{\max}) = \text{TRUE}$  then
6:       if  $h(k) \neq d$  then
7:         Add entry  $(k, d)$  to  $A'$ 
8:       Update  $\hat{L}(d)$ ; remove  $k$  from  $\mathcal{C}$ ; break;
9: return  $A'$ 
10: function  $\text{ADJUST}(k, d, \mathcal{C}, \theta_{\max})$ 
11:    $L_{\max} \leftarrow (1 + \theta_{\max})\bar{L}_{i-1}$ 
12:   if  $L_{i-1}(d) + c_{i-1}(k) < L_{\max}$  then
13:     return TRUE
14:   else if  $\exists \mathcal{E}$  selected by  $\psi$  and satisfying (i)-(iii) then
15:     foreach  $k \in \mathcal{E}$  do
16:       Disassociate  $k$  from  $d$ 
17:       Add  $k$  to  $\mathcal{C}$ 
18:     return TRUE
19:   else
20:     return FALSE

```

---

#### A. Least-Load Fit Decreasing (LLFD)

In this part of the section, we introduce *Least-Load Fit Decreasing* (LLFD) subroutine, which will be applied by all the proposed algorithms in Phase III, based on the idea of prioritizing keys with larger workloads. The design of LLFD is motivated by the classic First Fit Decreasing (FFD) used in conventional bin packing algorithms. The pseudo codes of LLFD are listed in Algorithm 1.

Generally speaking, LLFD sorts the keys in the candidate set in a non-increasing order of their computation costs and iteratively assigns the keys to task instances, such that (i) it generates the least total workload (Line 4); and (ii) it tries to adjust the key assignment, if the new destination task instance is overloaded after the migration (Line 5). If such key-to-instance pair is inconsistent with default mapping by



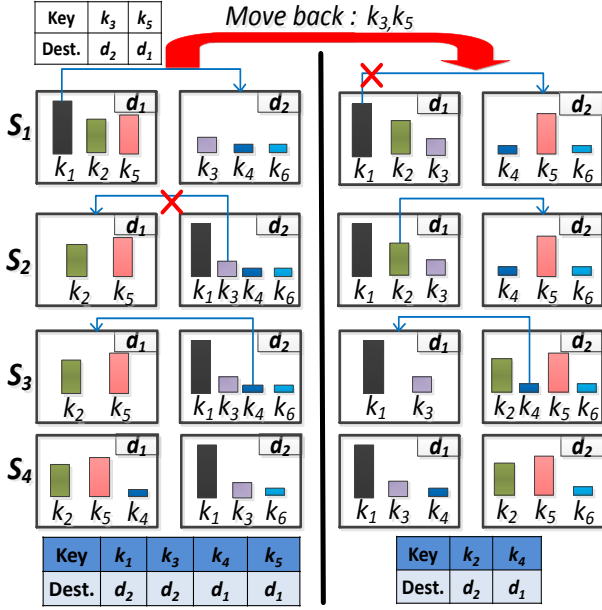


Fig. 4. Running examples for LLFD and MinTable with the constraints that the routing table size is not more than 2 and all instances should be absolute balance. The heights of the bars indicate the workloads of the corresponding keys. Each  $S_j$  with  $j = \{1, 2, 3, 4\}$  is a running step in the algorithms. The original routing table is at top of the figure, and the result routing tables are listed at the bottom.

hashing (Line 6), an entry  $(k, d)$  is then added to the routing table  $A$  (Line 7). After each iteration, LLFD updates the total workload of the corresponding instance  $d$  and removes  $k$  from the candidate set (Line 8). The iteration stops and returns the result routing table, when the candidate set turns empty (Line 9).

Basically, the algorithm moves the “heaviest” key to the task instance with minimal workload so far, which may generate another overloaded task instance (referred as “re-overloading” problem), if this key is associated with extremely heavy cost. Consider the toy example on the left side of Fig. 4. There are two instances:  $d_1$  is responsible for keys  $k_1, k_2$  and  $k_5$  with costs 7, 4 and 5 respectively, generating  $L(d_1) = 16$ , and  $d_2$  is associated with keys  $k_3, k_4$  and  $k_6$  with cost 2, 1 and 1 respectively, generating  $L(d_2) = 4$ . Suppose  $\theta_{\max} = 0$ , meaning that the total workloads on both instances are required to be equal (i.e., average workload  $\bar{L} = 10$ ). It is clear that  $d_1$  is overloaded and  $k_1$ , which incurs the largest computation cost, is expected to be disassociated from  $d_1$ . Although  $L(d_1)$  decreases to 9, it is still larger than  $L(d_2)$ . Based on the workflow of LLFD,  $k_1$  is assigned to  $d_2$ , only to overload  $d_2$  as a consequence. To tackle the problem, we add a new function, called *Adjust*, to avoid the happening of such conflicts.

Specifically, if re-overloading does not happen after an assignment, i.e.,  $L_{i-1}(d) + c_{i-1}(k) \leq L_{\max} = (1 + \theta_{\max})\bar{L}_{i-1}$ , this assignment is acceptable and *Adjust* immediately returns a TRUE (Lines 12-13). Otherwise (Lines 14-20), *Adjust* attempts to construct a nonempty key set (called *exchangeable* set and denoted by  $\mathcal{E}$ ), by applying the selection criteria  $\psi$  (e.g., highest workload first). The *exchangeable* set must satisfy the following three conditions: (i)  $\mathcal{E} \subseteq \{k' | F(k') = d\}$ ; (ii)  $\forall k' \in \mathcal{E}, c_{i-1}(k') < c_{i-1}(k)$ ; and (iii)  $L_{i-1}(d) + c_{i-1}(k) - \sum_{k' \in \mathcal{E}} c_{i-1}(k') \leq L_{\max}$ . Basically, (i) means that only keys

originally associated with  $d$  are selected for disassociation. (ii) tries not to choose a key with larger computation workload for disassociation, ensuring the decrease of the total workloads on instance  $d$ . Finally, (iii) ensures that instance  $d$  does not become overloaded, after the assignment (Lines 15-17).

Recall the running example in which LLFD tries to assign  $k_1$  to  $d_2$ , which makes  $d_2$  overloaded. A TRUE is returned by *Adjust* because there exists an  $\mathcal{E} = \{k_3\}$  satisfying constraints (i) - (iii). Therefore,  $k_1$  is assigned to  $d_2$ , while  $k_3$  is disassociated from  $d_2$  and put into  $\mathcal{C}$ . Next, LLFD attempts to assign  $k_3$  to  $d_1$ , because  $d_1$  has less total workload at this moment. However, a FALSE (a red cross shown on left side of  $S_2$  in Fig. 4) is returned by *Adjust* because overloading occurs (since  $L(d_1) + c(k_3) = 11 > L_{\max}$ ) and no valid  $\mathcal{E}$  exists, when neither of the two keys associated with  $d_1$  ( $k_2$  and  $k_5$ ) has smaller computation workload than that of  $k_3$ , violating constraint (ii). After this failure, LLFD is forced to consider another option, by keeping  $k_3$  to  $d_2$ . Luckily, a TRUE is returned this time, because a valid *exchangeable* set  $\mathcal{E} = \{k_4\}$  exists. After disassociating  $k_4$  from  $d_2$  and putting it into  $\mathcal{C}$ ,  $d_2$  is responsible for  $k_1, k_3$  and  $k_6$  only, the keys with  $d_1$  remains unchanged, and  $k_4$  is now in  $\mathcal{C}$ . The algorithm does not terminate until  $\mathcal{C}$  becomes empty, after  $k_4$  is assigned to  $d_1$ , finally reaching perfect balance at  $L(d_1) = L(d_2) = 10$ .

In the following, we present formal analysis on the robustness and soundness of LLFD on basic load balancing problem, with proofs available in Appendix. A.

**Theorem 1:** If there is a solution for absolute load balancing, LLFD always finds a solution resulting with balancing indicator  $\theta_i(d, F)$  no worse than  $\frac{1}{3}(1 - \frac{1}{N_D})$  for any task instance  $d_i$ .

## B. MinTable and MinMig Heuristics

The general workflow described above is essentially effective in guaranteeing load balance constraints, e.g., the LLFD sub-procedure. To address the optimizations on routing table minimization and migration cost minimization, we discuss two heuristics, namely MinTable and MinMig in this part of the section.

### algorithm 2 MinTable Algorithm

- 1: Phase I: *Move back* all keys in  $A$ .
- 2:  $\psi \leftarrow$  highest computation cost  $c(k)$  first
- 3: Phase II: According to  $\psi$ , select and disassociate keys from each of the overloaded instances, put them into  $\mathcal{C}$
- 4: Phase III:  $A' \leftarrow$  LLFD ( $\mathcal{C}, \mathcal{D}, \theta_{\max}, \psi$ )
- 5: **return**  $A'$

The pseudocodes of MinTable is shown in Algorithm 2. In order to minimize routing table size, in Phase I, all entries in routing table  $A$  are erased. The highest computation workload first criterion, which emphasizes on the computation cost, is used for the second and third phases, so that minimal number of entries are added into the new routing table  $A'$  during the key re-assignment and load rebalance process.

The two toy examples in Fig. 4 demonstrate how MinTable helps to achieve a smaller routing table while keeping load balance constraints fulfilled. The example on left side of Fig. 4

initially has two entries in routing table, i.e.,  $(k_3, d_2)$  and  $(k_5, d_1)$ . LLFD is directly applied to achieve absolute load balance  $L(d_1) = L(d_2)$ , but resulting in a routing table with four entries at the end. In contrast, before applying LLFD, the example on right side of Fig. 4 moves back  $k_3$  and  $k_5$  (i.e., cleaning the routing table). Finally, it results in a routing table with only two entries. The pseudo code of MinMig is shown in Algorithm 3. Although the removal of keys from the routing is *virtual* only, it increases the possibility of key migrations. Therefore, there is no cleaning run in the first phase at all.

---

**algorithm 3** MinMig Algorithm

---

- 1: Phase I: Do nothing.
  - 2:  $\psi \leftarrow$  largest  $\gamma_i(k, w)$  first, where  $\gamma_i(k, w) = \frac{c_i(k)^\beta}{S_i(k, w)}$
  - 3: Phase II: According to  $\psi$ , select and disassociate keys from each of the overloaded instances, put them into  $\mathcal{C}$
  - 4: Phase III:  $A' \leftarrow$  LLFD  $(\mathcal{C}, \mathcal{D}, \theta_{max}, \psi)$
  - 5: **return**  $A'$
- 

To characterize both computation and migration cost, we propose the *migration priority index* for each key, defined as  $\gamma_i(k, w) = c_i(k)^\beta S_i(k, w)^{-1}$ . Its physical meaning is straightforward, that is, a key with larger computation cost per unit memory consumption has the higher priority to be migrated. The weight scaling factor  $\beta$  is used to balance the weights between these two factors under consideration. Consider  $k_1$  and  $k_2$  in Fig. 4 and assume window  $w = 1$ . We have  $c(k_1) = S(k_1, w) = 7$  and  $c(k_2) = S(k_2, w) = 4$ . If we give equal weights to both  $c(k)$  and  $S(k, w)$ , i.e.,  $\beta = 1$ , then  $\gamma(k_1, w) = \gamma(k_2, w) = 1$ . When we assign more importance to migration cost, i.e.,  $\beta = 0.5$ ,  $k_2$  gains higher priority for migration. In addition,  $\beta$  also affects the size of the result routing table, i.e., the larger  $\beta$ , the smaller size of routing table, which will be shown in the experiment results in Appendix A. The largest  $\gamma_i(k, w)$  first criterion, which is aware of both computation and migration cost, is used during both key re-assignment (Phase II) and load balance process (Phases III), in order to minimize the bandwidth used to migrate the states of keys (e.g., the tuples in sliding window for join operator).

### C. Mixed Algorithm

Based on the discussion on the heuristics, we discover that there are tradeoffs between routing table minimization and migration cost minimization. Therefore, we propose a mixed algorithm to intelligently combine the two heuristics MinTable and MinMig, in order to produce the best-effort solutions towards our target optimization in Eq. 3.

The basic idea is to properly mix MinTable (Phase I) and MinMig (Phases II and III). In the first phase, the mixed strategy *moves back*  $n$  keys, which are selected from  $A$ , based on the smallest memory consumption  $S_{i-1}(k, w)$  first criteria. The rest two phases simply follow the procedure of MinMig, in which the largest  $\gamma_i(k, w)$  first criteria is used to initialize candidate key set  $\mathcal{C}$  and applied by LLFD in the last phase. For the Mixed algorithm, the most challenging problem is how to pick up the number of keys for back moves, i.e.,  $n \in [0, N_A]$  during the cleaning phase. Actually, MinTable and MinMig works on two extremes of the spectrum in this step, such that  $n = N_A$  in MinTable and  $n = 0$  in MinMig.

---

**algorithm 4** Mixed Algorithm

---

- 1:  $\eta \leftarrow$  smallest memory consumption  $S_i(k, w)$  first.
  - 2:  $\psi \leftarrow$  largest  $\gamma_i(k, w)$  first, where  $\gamma_i(k, w) = \frac{c_i(k)^\beta}{S_i(k, w)}$
  - 3:  $n \leftarrow 0$
  - 4:  $A_{backup} \leftarrow A$
  - 5: **do**
  - 6:    $A \leftarrow A_{backup}$
  - 7:   Phase I: According to  $\eta$ , select  $n$  keys from  $A$  and *move back* them
  - 8:   Phase II: According to  $\psi$ , select and disassociate keys from each of the overloaded instances, put them into  $\mathcal{C}$
  - 9:   Phase III:  $A' \leftarrow$  LLFD  $(\mathcal{C}, \mathcal{D}, \theta_{max}, \psi)$
  - 10:    $n = N_{A'} - A_{max}$
  - 11: **while**  $n > 0$
  - 12: **return**  $A'$
- 

Obviously, brute force search (named as *Mixed<sub>BF</sub>*) could be applied to try with every possible  $n = 1, 2, \dots, N_A$ , with the optimal  $n^*$  returned after evaluating the solution with every  $n$ . Alternatively, we propose a faster heuristic in Algorithm 4. It only tries a small number of values, which are the amount of table entries overused in the last trial (Line 10). The trial starts from  $n = 0$  (Line 3, same as MinMig), and stops when it results in an updated  $A'$  of acceptable size, i.e.,  $N_{A'} \leq A_{max}$  (Lines 11-12). Note that the efficiency of the algorithm is much better than *Mixed<sub>BF</sub>*, although it may not always find the optimal  $n^*$  as *Mixed<sub>BF</sub>* does. Obviously, the size of the result routing table by the mixed algorithm is no smaller than that of MinTable approach. Similarly, the migration cost of the result assignment function is no smaller than that of the MinMig approach. However, mixed algorithm is capable of hitting good balance between the heuristics, as is proved in our empirical evaluations. Furthermore, the balance status generated by Mixed as the following theorem (proof in Appendix. A):

*Theorem 2:* Balance status generated by Mixed is not worse than that generated by LLFD.

## IV. IMPLEMENTATION OPTIMIZATIONS

The overall working mechanism of the rebalance control component, as is implemented in our distributed stream processing engine, is illustrated in Fig. 5. In the figure, each operation step is numbered to indicate the order of their execution.

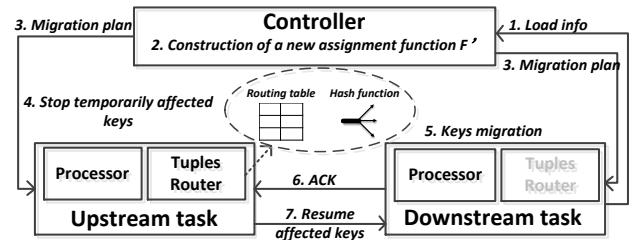


Fig. 5. Overall workflow.

At the end of each time interval (e.g., 10 seconds as the setting in our experiments), the instances of an operator report the statistical information collected during the past interval to a

controller module (step 1). The information from each instance  $d$  includes the computation cost  $c_{i-1}(k)$  and window-based memory consumption  $S_{i-1}(k, w)$  of each key assigned to it. On receiving the reporting information, the controller starts the optimization procedure (step 2) introduced in Section III. It first evaluates the degree of workload imbalance among the instances and decides whether or not to trigger the construction of a new assignment function  $F'$  to replace the existing  $F$ . If the system identifies load imbalance, it starts to execute Mixed algorithm (Algorithm 4) to generate new  $A'$  and  $F'$ .

After calculating the keys in  $\Delta(F, F')$  for migration, the controller broadcasts both  $F'$  and  $\Delta(F, F')$ , together with a **Pause** signal to all the instances of upstream operator for them to update the obsolete  $F$ , and temporarily stop sending (but caching locally) data with keys in  $\Delta(F, F')$  (steps 3 and 4). Meanwhile, the controller notifies the corresponding downstream instances (step 3).

Finally, the instances of downstream operator begin migrating the states of keys after the notification from the controller (step 5) and acknowledge the controller when migration is completed (step 6). As soon as the controller receives all the acknowledgments, it sends out a **Resume** signal to all instances of the upstream operator, ordering the tasks to start sending data with keys in  $\Delta(F, F')$ , since all the downstream instances are equipped with the new assignment function (step 7). It is worth noting that during the key state migration, there is no interruption of normal processing on the data with keys not covered by  $\Delta(F, F')$ .

One potential problem in the workflow above is the cost of transmitting statistical information with the keys in step 1, which could easily contain millions of unique keys in real application domains. The huge size of the key domain may degenerate the scalability of the algorithms, on growing computational complexity and memory consumption for these metrics. To alleviate the transmission problem, we propose a compact representation for the keys with acceptable information loss for the algorithms.

The basic idea is to merge the keys with common characteristics and represent them by a single record in the statistical data structure. To accomplish this goal, we design a new 6-dimensional vector structure for the statistical information,  $(d', d, d^h, v_c, v_S, \#)$ , in which  $d'$  denotes the instance to which a key will be assigned next;  $d$  is the instance with which the keys are currently associated during the reporting period (i.e.,  $d = F(k)$ );  $d^h$  is the instance assigned by the hash function (i.e.,  $d^h = h(k)$ );  $v_c$  denotes the value of computation workload;  $v_S$  is the value of window-based memory consumption; and  $\#(> 0)$  is the number of keys satisfying these five conditions. For example, a vector  $(d_1, d_2, d_1, 4, 4, 2)$  indicates that there are two keys with computation workload 4 and memory consumption 4. They are currently associated with instance  $d_2$ , and the instance suggested by hash function is  $d_1$ , indicating that the routing table  $A$  at the upstream operator must contain an entry for them. Finally, the record also implies that they will be assigned to  $d_1$ , meaning that the entry for them in  $A$  is deleted and the *move back* operation is executed on these two keys.

By employing this compact representation, the whole key space  $\mathcal{K}$  is transformed to the 6-dimensional vector space (de-

noted by  $\mathcal{K}^c$ ). The upper bound on the size of the vector space is approximately  $K^c = |\mathcal{K}^c| = O(N_D^3 \times |c(k)| \times |S(k, w)|)$ , where  $N_D$  is the number of downstream instances, which is usually a small integer;  $|c(k)|$  and  $|S(k, w)|$  represent the total numbers of distinct values on computation workload and memory consumption in current sliding window, respectively.

#### A. Mixed Algorithm over Compact Representations

Apparently, the compact representation brings significant benefits, by reducing both time and space complexity of the Mixed algorithm (and MinTable and MinMig as well) proposed in Section III. In the following, we briefly describe how Mixed algorithm is revised based on the compact representation. To make a clear description, let us revisit the Mixed algorithm and look into the steps using the compact representations.

**Phase I (Cleaning):** According to the smallest  $S_{i-1}(k, w)$  first criterion, the Mixed algorithm selects  $n$  keys from  $A$  and moves them back to original instance based on the hash function. In the compact representation, the adapted Mixed algorithm does not target on any individual key, but the 6-dimensional vectors. In result, a *back-move* of keys is equivalent to modifying the value of  $d'$  to be the same as  $d^h$  of the selected vectors. The vector  $(d_1, d_2, d_1, 4, 4, 2)$  mentioned above presents an example back-move of a number of keys.

**Phase II (Preparing):** When investigating the workloads of a particular instance, say  $d_1$ , the adapted Mixed algorithm calculates the weighted sum of  $v_c \times \#$  of all records containing  $d' = d_1$ . If we have two records  $(d_1, d_2, d_1, 4, 4, 2)$  and  $(d_1, d_2, d_2, 8, 8, 1)$ , for example, the total workload with respect to the keys is estimated as 16. Next, when the adapted Mixed algorithm needs to select keys from an overloaded instance and puts them into the candidate set  $\mathcal{C}$ , the algorithm again targets on those vectors in compact representations, and simply replaces the value of  $d'$  with a *nil*, indicating a virtual removal of the keys linked to the vector. For example, if  $(d_2, d_2, d_1, 4, 4, 2)$  is disassociated from  $d_2$  and put into  $\mathcal{C}$ , the adapted Mixed algorithm finally rewrites the record as  $(nil, d_2, d_1, 4, 4, 2)$ . Note that when a record is added into  $\mathcal{C}$ , i.e., containing  $d' = nil$ , it is likely that there already exists a record in  $\mathcal{C}$  with exactly the same values on  $d, d^h, v_c$  and  $v_S$ . According to the definition on the uniqueness of the compact representation, these two records need to be merged by summing on the number field ‘#’.

**Phase III (Assigning):** The similar adaptation in Phase I and II is also applied to LLFD, with only one exception that the expected routing table  $A'$  can not be directly derived but rather indirectly calculated. This is because the final results returned by the adapted LLFD is still in a compact form, as a 6-dimensional tuple. In order to derive  $A'$  and  $F'$ , a series of additional actions are taken, including (i) picking up those needing migration from the records returned by adapted LLFD, i.e., tuples with  $d' \neq d$ ; (ii) selecting keys originally associated with instance  $d$  and computation cost at  $v_c$ , according to the selection criteria  $\psi$  and based on the original complete statistical information of keys collected by the controller; and (iii) adding them to the key migration set  $\Delta(F, F')$ . Finally, the adapted algorithm returns the final result, with  $F'$  induced by combining  $F$  and  $\Delta(F, F')$ , and  $A'$  derived with  $F'$  together with  $h(k)$ .



### B. Discretization on $v_c$ and $v_S$

As is emphasized above, the size of the 6-dimensional vector space  $K^c = O(N_D^3 \times |c(k)| \times |S(k, w)|)$  depends on  $|c(k)|$  and  $|S(k, w)|$ . In practice, the values of computation cost and memory consumption could be highly diversified, leading to large  $|c(k)|$  and  $|S(k, w)|$ , and consequently huge vector space  $K^c$ . It is thus necessary to properly discretize the candidate values used in  $c(k)$  and  $S(k, w)$ , in order to control the complexity blown by  $|c(k)|$  and  $|S(k, w)|$ .

Value discretization can be done in a straightforward way. However, an over-simplistic approach could cause huge deviations on the approximate values from the real ones. Such deviations may affect the accuracy of workload estimation and jeopardize the usefulness of the migration component. For instance, assume that there are 10 keys with computation costs  $c(k_1) = 8, c(k_2) = 6, c(k_3) = 3, c(k_4) = c(k_5) = 2$  and  $c(k_6) = \dots = c(k_{10}) = 1$ . If a simple piecewise constant function is used, i.e.,  $\xi(x) = 2$  when  $x \in [1, 3]$ ,  $\xi(x) = 5$  when  $x \in [4, 6]$ ,  $\xi(x) = 8$  when  $x \in [7, 9]$ , and 0 otherwise. Despite of a very small  $|c(k)| = 2$ , the total deviation, following the formula below, caused by the approximation is fairly large:

$$|\delta| = \left| \sum_{i=1}^{10} \delta_i \right| = \left| \sum_{i=1}^{10} c(k_i) - \xi(c(k_i)) \right|.$$

Fig. 6(a) illustrates how this simple piecewise constant function fails and the deviation  $\delta_i$  with respect to each value point.

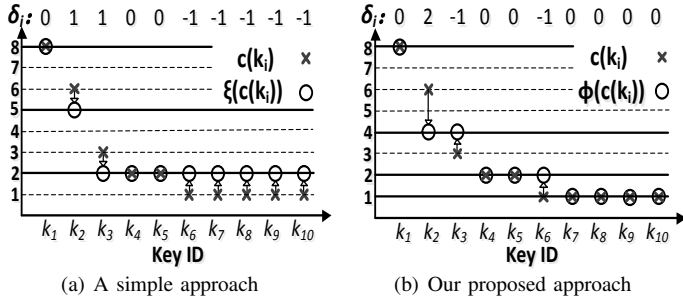


Fig. 6. An example of comparing the simple piecewise discretization function (a) and our proposed approach (b).

To tackle the problem, we propose an improved discretization approach, denoted by  $\phi(x)$ , which involves two steps. In the first step, it generates a finite number of representative values. Secondly, instead of using the nearest representative for each value independently, our approach constructs the discretized values in a more holistic manner. Assume the input value is a series of  $n$  numbers in a non-increasing order by their values, of which the smallest is at least 1 (after normalization), i.e.,  $x_1, x_2, \dots, x_n, \forall i, i' \in [1, n], i < i', x_i \geq x_{i'} \geq 1$ .

In the first step, a simple method (half-linear-half-exponential, HLHE) with a parameter  $R$  (called the degree of discretization) is applied to determine the representative values, where we require  $R = 2^r, r = 0, 1, 2, \dots$ . Therefore, a total number of

$$m = r + \left\lfloor \frac{\max(x_i)}{R} \right\rfloor = r + s$$

representative values are generated and reorganized as a strictly decreasing series,  $y_1, y_2, \dots, y_m$ , where  $y_1 = s \times R, y_2 = (s-1) \times R, \dots, y_s = R$  (the linear part), and  $y_{s+1} = R/2 = 2^{r-1}, y_{s+2} = 2^{r-2}, \dots, y_{m-1} = 2, y_m = 1$  (the exponential part).

In the second stage, a greedy method is applied to finalize the discretization by adopting an optimization framework. The basic principle is to minimize the accumulated error of all values, such that the sum over an arbitrary set of approximate values tend to be an accurate estimation to the sum over original values. Specifically, for each  $x_i < y_1$ , two representative values  $j \in [2, m]$  that  $y_{j-1} > x_i \geq y_j$  can be used to approximate  $x_i$ . We define such  $y_{j-1}$  and  $y_j$  as candidate representative values for  $x_i$ . For the remaining ( $x_i \geq y_1$ ), they only have one candidate representative value, which is  $y_1$ . For each  $x_i$ , one of the candidate representative values is chosen when there are two options, denoted by  $\phi(x_i)$ , so that the total deviation  $|\delta|$  is minimized. In particular, if the current accumulated deviation is positive,  $x_i$  is represented by the larger value  $y_{j-1}$  in order to cancel the over-counting. Otherwise,  $x_i$  chooses the representative value  $y_j$ .

In the example of Fig. 6(b), we let  $r = 2$  and  $R = 4$ , thus  $m = 2 + \frac{8}{4} = 4$ . There are four representative values, e.g.,  $y_1 = 8, y_2 = 4, y_3 = 2, y_4 = 1$ . At the time  $k_3$ , whose  $c(k_3) = 3$ , is processed, the two representative values for it are  $y_2 = 4$  and  $y_3 = 2$  respectively. Since the accumulated deviation caused by  $k_1$  and  $k_2$  equals to 2, we have  $\phi(c(k_3)) = y_2 = 4$ . This results in a reduction on the accumulated deviation by 1. When our proposed approach terminates, according to Fig. 6(b), the total deviation is zero, while the simple piecewise constant function generates a total deviation at  $|\delta| = 3$ .

Based on discussion above, we have the following theorem:

**Theorem 3:** The value discretization always can be done perfectly ( $|\delta| \sim 0$ ) by the above two steps.

*Proof:* Due to data skew, the number of small load key is always more than that of big load key. Furthermore, the smallest representative values  $r = 0, 1, 2$  will not cause the accumulated error of values. Then, the value discretization always can be done perfectly ( $|\delta| \sim 0$ ) by the above two steps. ■

## V. EVALUATIONS

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  $N_D$  and routing table size  $N_A$ . To collect the workload measurements, we add a load reporting module into the processing logics when implementing them in *Storm*'s topologies. Migration and scheduling algorithms are injected into the codes of *controllers* in *Storm* to enable automatic workload redistribution. We use the consistent hashing [14] as our basic hash function and configure the parallelism of spout at 10. By controlling the latency on tuple processing, we force the distributed system to reach a saturation point of CPU resource for the  $N_D$  number of processing tasks with the requirement of absolute load balancing ( $\theta_{max} = 0$ ). We show the results are averages of 5 runs. The *Storm* system (in version 0.9.3) is deployed on a 21-instance HP blade cluster with CentOS 6.5 operating system.

Each instance in the cluster is equipped with two Intel Xeon processors (E5335 at 2.00GHz) having four cores and 16GB RAM. Each core is exclusively bound with a worker thread during our experiments.

TABLE II. PARAMETER SETTINGS

	Range	Description
$K$	$[5 \cdot 10^3, 10^4, 10^5, \mathbf{10^6}]$	Size of key domain
$z$	$[0, \dots, \mathbf{0.85}, \dots, 1.0]$	Distribution skewness
$f$	$[0, \dots, \mathbf{1.0}, \dots, 2.0]$	Fluctuation rate
$\theta_{\max}$	$[0, \dots, \mathbf{0.08}, \dots, 1.0]$	Tolerance on load imbalance
$\beta$	$[1, \dots, \mathbf{1.5}, \dots, 2.0]$	Migration selection factor
$r$	$[0, 1, 2, \mathbf{3}, 4, 5, 6, 7, 8]$	Level partition distance
$w$	$[1, \mathbf{5}, 10, 15, 20]$	Time interval
$N_D$	$[1, 5, 10, \mathbf{15}, 20, \dots, 40]$	Number of task instances
$N_A$	$[0, \dots, \mathbf{3 \cdot 10^3}, \dots, 5 \cdot 10^4]$	Size of the routing table

**Synthetic Data:** Our synthetic workload generator creates snapshots of tuples for discrete time intervals from an integer key domain  $K$ . The tuples follow Zipf distributions controlled by skewness parameter  $z$ , by using the popular generation tool available in Apache project. We use parameter  $f$  to control the rate of distribution fluctuation across time intervals. At the beginning of a new interval, our generator keeps swapping frequencies between keys from different task instances until the change on workload is significant enough, i.e.,  $\frac{|L_i(d) - L_{i-1}(d)|}{L_i(d)} \geq f$ . Parameter  $\theta_{\max}$  is defined as the tolerance of load imbalance, measured as the ratio of maximal workload to minimal workload among the task instances. Our algorithm *Mixed* is controlled by two more parameters  $\beta$  and  $r$ , as defined in previous sections. The range of the parameters tested in the experiments are summarized in Tab. II, with default values highlighted in bold font. We also employ *TPC-H* tool *DBGen* [2] to generate a synthetic warehousing workload. And we revise *Q5* in *TPC-H* into a continuous query over sliding window as our testing target, because *Q5* includes all primitive database operations.

**Real-World Data:** We also use two real workloads in the experiments. The first *Social* workload includes 5-day feeds from a popular microblog service, in which each feed is regarded as a tuple with words as its keys. There are over 5,000,000 tuples covering 180,000 topic words as the keys. The second workload includes 3-day *Stock* exchange records, consisting of over 6,000,000 tuples with 1,036 unique keys (Stock ID) for stock transactions. For both datasets, we take each day as a time interval, so the workload inside one window size consists of the tuples in the last 24 hours. We run word count topology on *Social* data, which continuously maintaining current tuples in memory and updating the appearance frequency of topic words in social media feeds. We run self-join on *Stock* data over sliding window, used to find potential high-frequency players with dense buying and selling behavior. These two workloads evolve in completely different patterns. To be specific, the word frequency in *Social* data usually changes slowly, while *Stock* data contains more abrupt and unexpected bursts on certain keys.

**Baseline Approaches:** We use *Mixed* to denote our proposed algorithm mixing two types of heuristics. We also use *Mixed<sub>BF</sub>* to denote the brute force version of *Mixed* method, which completely rebuilds the routing table from

scratch at each scheduling point. We use *MinTable* to denote the algorithm always trying to find migration plan generating minimal routing table. Finally, we also include *Readj* and *PKG* as baseline approaches, which are known as state-of-the-art solutions in the literature. *Readj* is designed to minimize the load of restoring the keys based on the hash function, implemented by key rerouting over the keys with maximal workload. The migration plan of keys for load balance is generated by pairing tasks and keys. For each task-key pair, their algorithm considers all possible swaps to find the best move alleviating the workload imbalance. In *Readj*,  $\sigma$  is a configurable parameter, deciding which keys should take part in action of swap and move. Given a smaller  $\sigma$ , *Readj* tend to track more candidate keys and thus finding better migration plans. In order to make fair comparison, in each of the experiment, we run *Readj* with different  $\sigma$ s and only report the best result from all attempts. *PKG* [21] is a load balancing method without migration at runtime. It balances the workload of tasks by splitting keys into smaller granularity and distributing them to different tasks based on randomly generated plan. Here, we only use *PKG* approach for simple aggregation processing in the experiments, because it does not support complex stateful operations, such as join. Due to the unique strategy used by *PKG*, aggregation topologies run on *PKG* must contain a special downstream operator in the topology, which is used to collect and merge partial results with respect to every key, from two independent workers in the upstream operator. Moreover, in the open source version of *PKG*<sup>1</sup>, there is a parameter  $p$  indicating the time interval between two consecutive result merging. After careful investigation with experiments, we find a larger  $p$  prolongs the response time of tuple processing, reduces the additional computation cost and limits the maximal number of live tuples (known as maximal pending tuples in *Storm*) under processing in the system. We finally chose  $p$  at 10 milliseconds and set maximal pending tuples at 50, which generally maximizes the throughput of *PKG* in all settings. Note that we do not include *LLFD* and *MinMig* algorithms in the experiments, because both of them can not control the size of routing tables, therefore blowing off the memory space of the tasks in some cases.

**Evaluation Metrics:** In the experiments, we report the following metrics. Workload skewness (i.e.,  $\frac{\max L(d)}{\bar{L}}$ ), is the ratio of maximal workload on individual task instance to the average workload. Migration cost reveals the percentage of states associated with the keys involved in migration over the states maintained by all task instances. Throughput is the average number of tuples the system processes in unit second. Average generation time is the average time spent on the generation of migration plan in *Storm* controller. Finally, processing latency is the average latency of individual tuples, based on the statistics collected by *Storm* itself. In the rest of the section, we report the average values for these metrics over complete processing procedure, as well as the minimal and maximal values when applicable, to demonstrate the stability of different balance processing algorithms.

**Load Skewness Phenomenon:** To understand the phenomenon of workload skewness with traditional hash-based mechanism, we report the workload imbalance phenomenon

<sup>1</sup><https://github.com/gdfm/partial-key-grouping>

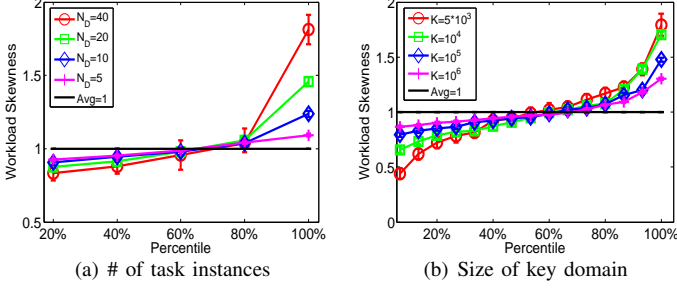


Fig. 7. Cumulative distribution of workload skewness under hash-based scheme.

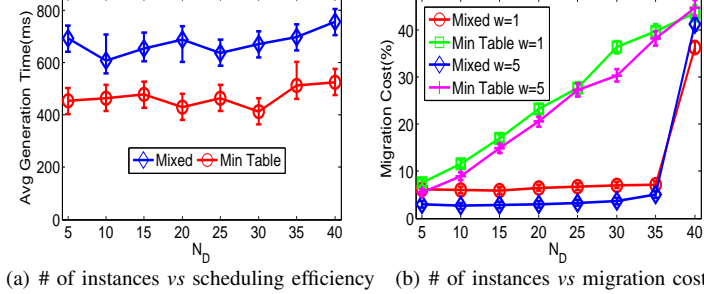


Fig. 8. Performance with varying number of task instances.

on the task instances by changing the number of task instances and the size of key domain, respectively. The results of load imbalance in Fig. 7 are presented as the cumulative distribution of average workload among the task instances over 50 time intervals. Fig. 7(a) implies that the skewness grows when increasing the number of task instances. When there are 40 instances (i.e.,  $N_D = 40$ ), the maximal workload at 100% percentile is almost 2.5 times larger than the minimal workload. Fig. 7(b) shows that the workload imbalance is also highly relevant to the size of key domain. When there are more keys in the domain, the hash function generates more balanced workload assignment. In Fig. 7(b), the maximal workload for  $K = 5,000$  is around 4 times larger than the minimal one and is much larger than the maximal load under larger key domain size (e.g.,  $K = 1,000,000$ ). Therefore, workload imbalance for intra-operator parallelism is a serious problem and cannot be easily solved by randomized hash functions.

**Impact of Algorithm Parameters:** We test the algorithm parameters on synthetic datasets using two window sizes (i.e.,  $w = 1$  and  $w = 5$ ), in order to understand their impacts for short and long term aggregation over stream data. When  $w = 1$ , migration decisions are made based on the current stateful and instantaneous workload. When  $w = 5$ , more state information in the last five intervals are included in the decision making procedure.

Although the increase on  $N_D$  produces more workload imbalance, our migration algorithm *Mixed* performs well, by generating excellent migration plan, as shown in Fig. 8. *Mixed* costs a little additional overhead over *MinTable* algorithm for balancing, but its migration cost is much lower than *MinTable* when  $N_D \leq 35$  for both  $w = 1$  and  $w = 5$ , as presented in Fig. 8(b). The cleaning step in *MinTable* algorithm also leads to even higher skewness and much more migration cost in order to achieve load balancing. When  $w = 5$ , *Mixed* keeps more

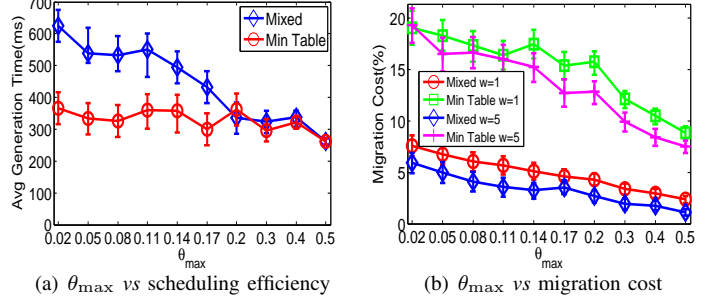


Fig. 9. Performance with varying  $\theta_{\max}$ .

historical tuples which can be used as the migration candidates. This makes the migration easier and less expensive, when compared to the case with  $w = 1$ . When  $N_D > 35$ , however, the migration cost of *Mixed* jumps, almost reaching the cost of *MinTable* when  $N_D = 40$ . This is because the outcome of *Mixed* algorithm degenerates to that of *MinTable* algorithm, when the minimal routing table size needed for target load balancing exceeds the specified size of the table in the system.

Fig. 9 displays the efficiency of migration plan generation and the corresponding migration cost with varying workload balancing tolerance parameter  $\theta_{\max}$ . As expected, Migration scheduling runs faster on synthetic dataset with larger  $\theta_{\max}$  in Fig. 9(a). When  $\theta_{\max} \geq 0.2$ , the efficiency of *Mixed* catches that of *MinTable*. If stronger load balancing (i.e., smaller  $\theta_{\max}$ ) is specified, system pays more migration cost as shown in Fig. 9(b), basically due to more keys involved in migration. But *MinTable* incurs three times of the migration cost of *Mixed* under the same balance requirement. Even for strict  $\theta_{\max} = 0.02$  (almost absolutely balanced), the algorithm is capable of generating the migration plan within 1 second. Moreover, migration cost with larger window size (i.e.,  $w = 5$ ) shrinks, as the historical states provide more appropriate candidate keys for migration plan generation.

In Fig. 10, we report the results on varying key domain size  $K$ . By varying  $K$  from 5,000 to 1,000,000, *Mixed* spends more computation time on migration planning but incurs less migration cost than *MinTable*. As shown in Fig. 7(b), the smaller the key domain is, the more skewed the workload distribution will be. But our proposed solution *Mixed* shows stable performance, regardless of the domain size, based on the results in Fig. 10(a). In particular, migration cost decreases for both *MinTable* and *Mixed* algorithms, when the window size grows to  $w = 5$ .

In Sec. IV, we present the possibility of efficiency improvement by applying compact representation for key related information. In this group of experiments, we report the performance of this technique in Fig. 11, by varying the degree of discretization (i.e., the value of  $R$ ) on values of computation cost  $v_c$  and memory consumption  $v_S$ . Fig. 11(a) shows that discretization on both  $v_c$  and  $v_S$  is an important factor to the efficiency of migration scheduling. The average generation time of migration plan is quickly reduced when we allow the system to discretize the values at a finer granularity. Note that the point with label “Original Key Space” is the result without applying the compact representation on keys, while the point at  $R = 1$  is the case of the finest degree of

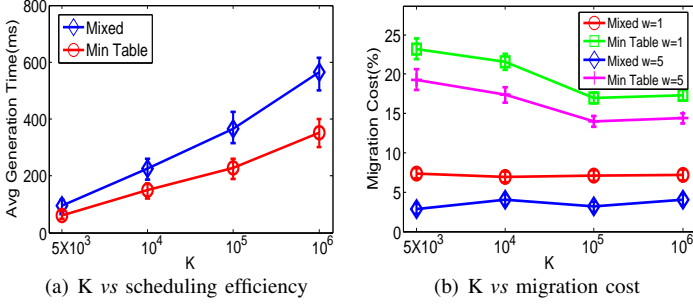


Fig. 10. Scheduling efficiency in terms of average generation time and migration cost under different key domain size, K

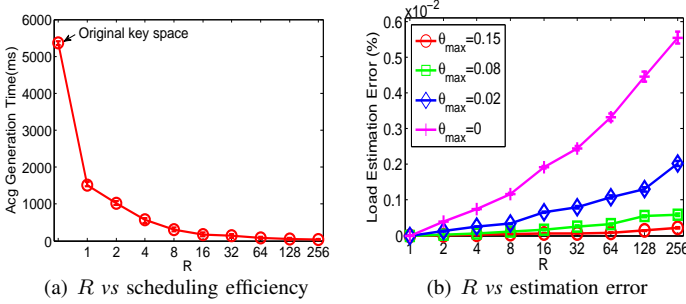


Fig. 11. Performance with varied degrees of discretization for partitioning granularity.

discretization on  $v_c$  and  $v_s$ . The efficiency is improved by an order of magnitude when  $R = 8$  (i.e., 0.6 second) compared to “Original Key Space” (i.e., 6 seconds). Although larger  $R$  leads to smaller  $|v_c|$ ,  $|v_s|$  and smaller  $K^c$  and makes the migration plan faster, the error on load estimation grows (i.e., the percentage of divergence between actual workload of a task instance and the estimated workload based on the discretized workload over the keys), as shown in Fig. 11(b), because the discretization generates inaccurate load approximation for the keys. However, such errors are no more than 1% in all cases, while the degree of discretization  $R$  varies from 1 to 256 as shown in Fig. 11(b).

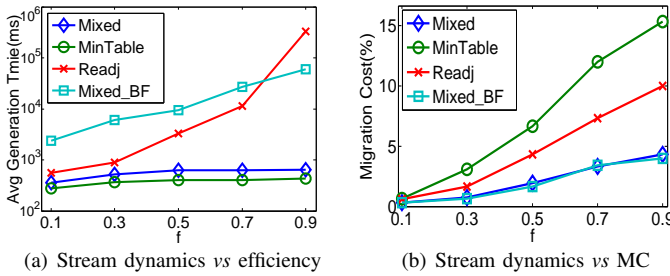


Fig. 12. Scheduling efficiency and migration cost with varying distribution change frequency.

Since *Readj* is the most similar technique to our proposal in the literature, we conduct a careful investigation on performance comparison to evaluate the effectiveness of our proposal. To optimize the performance of *Readj*, we adopt binary search to find the best  $\delta$  for *Readj*. Fig. 12 shows the performance on dynamic stream processing with imbalance tolerance  $\theta_{\max} = 0.08$ , by varying distribution change frequency  $f$ . When increasing  $f$ , *Readj* presents less

promising efficiency when generating migration plan, since it evaluates every pair of task instances and considers all possible movements across the instances. Instead, *Mixed* makes the migration plan based on heuristic information, which outperforms *Readj* by a large margin. The results also imply that brute force search with *MixedBF* is a poor option for migration scheduling. When variances occur more frequently (i.e., with a higher  $f$ ), migration cost of *Mixed* grows slower than that of *Readj*, while *MixedBF* performs similarly to *Mixed*.

**Throughput and Latency on Synthetic and Real Data :** In Fig. 13, we draw the theoretical limit of the performance with the line labeled as *Ideal*, which simply shuffles the workload regardless of the keys. Obviously, *Ideal* always generates a better throughput and lower processing latency than any key-aware scheduling, but cannot be used in stateful operators for aggregations. When varying the distribution change frequency  $f$ , both the throughput and latency of *Readj* change dramatically. In particular, *Readj* works well only in the case with less distribution variance (smaller  $f$ ). On the other hand, our *Mixed* algorithm always performs well, with performance very close to the optimal bound set by *Ideal*.

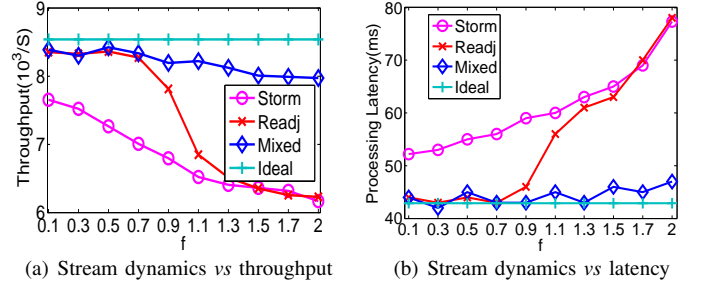


Fig. 13. Throughput and latency with varying distribution change frequency.

On *Social* data, we implement a simple word count topology on Storm, with upstream instances distributing tuples to downstream instances for store and aggregation on keywords. On *Stock* data, a self-join on the data over sliding window is implemented, which maintains the recent tuples based on the size of the window over intervals. The result throughputs are presented in Fig. 14. The most important observation is that the best throughput, on both of the workloads, is achieved by running *Mixed* with  $\theta_{\max} = 0.02$ , implying that strict load balancing is beneficial to system performance. *Mixed* also presents huge performance advantage over the other two approaches, with throughput about 2 times better than *Storm* and *Readj* at smaller  $\theta_{\max}$  in Fig. 14(b). The performance of *Readj* improves by relaxing the load balancing condition, catching up with the throughput of *Mixed* at  $\theta_{\max} = 0.3$  ( $\theta_{\max} = 0.15$  resp.) on *Social* (*Stock* resp.) This is because *Readj* works only when the system allows fairly imbalance among the computation tasks, for example  $\theta_{\max} = 0.3$ . *MinTable* does not care about migration cost and then it incurs larger migration volume, which reduces the throughput of system during the process of adjustment. *PKG* splits keys into smaller granularity and distributes them to different tasks selectively. Therefore, throughput of *PKG* is independent of the choice of  $\theta_{\max}$ , validated by the results in Fig. 14(a). The throughput of *PKG* is worse than *Mixed*, because its processing involves coordination between two operators. Despite of its excellent performance on load balancing, the overhead of partial result merging leads



to additional response time increase and overall processing throughput reduction. Overall, as shown in Fig. 14(a), when  $\theta_{max} = 0.02$ , our method outperform *PKG* on throughput by 10% and on response latency by 40%. Moreover, we emphasize that *PKG* cannot be used for complex processing logics, such as join, and therefore is not universally applicable to all stream processing jobs.

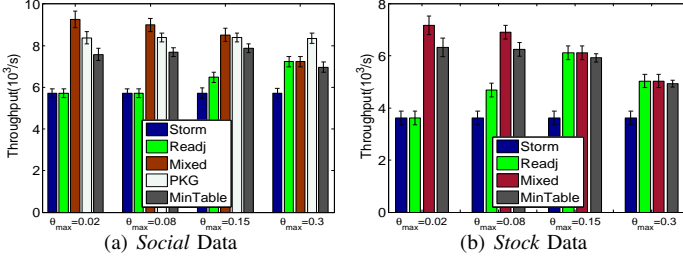


Fig. 14. Throughput on real data.

**Scalability on Real Data:** To better understand the performance of the approaches in action, we present the dynamics of the throughput over time on two real workloads, especially when the system scales out the resource by adding new computation resource to the operator. The results are available in Fig. 15. In order to test this kind of scale-out ability of different algorithms, we run the stream system to a balance status, and then add one more working thread (instance) to the system starting the balance processing algorithms. The results show that our method *Mixed* perfectly rebalances the system within a much shorter response time than that of *Readj*. Though *PKG* is  $\theta_{max}$  insensitive, it produces a lower throughput than *Mixed* while  $\theta_{max} = 0.1$ . As the explanation of Fig. 14(a), *PKG* needs to keep track of all the derived data from a spout until it receives ack response and this action exacerbates its processing latency. On *Social data* with  $\theta_{max} = 0.10$ , *Readj* takes at least 5 minutes to generate the migration plan for the new thread added to the system. Such a delay leads to huge resource waste, which is definitely undesirable to cloud-based streaming processing applications. Similar results are also observed on *Stock*. The quick response of *Mixed* makes it a much better option for real systems.

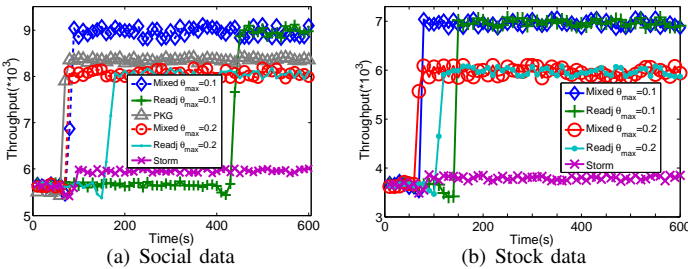


Fig. 15. Performance during system scale-out.

**Dynamics on TPC-H for Q5:** We use *DBGen* [2] to generate 1 GB TPC-H dataset by producing zipf skewness on foreign keys with  $z = 0.8$ . We run Q5 on the generated dataset for one hour and set window size as 5 minutes, since the join operations in Q5 are implemented by different processing operators. The data imbalance slows down the previous join operator (upstream instances) and suspends the processing on downstream join operators. This bad consequence of such suspension may be amplified with the growing number of

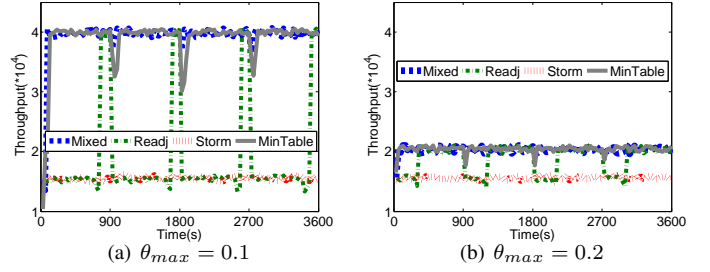


Fig. 16. Dynamic adjustment on TPC-H data for Q5.

task instances. In particular we test the effects by triggering the distribution change in every 15 minutes with  $f = 1$ . The results are shown in Fig. 16. Without any balancing strategy, *Storm* presents poor throughputs. *Mixed* is capable of balancing the workload in an efficient manner and achieving the best throughput under any balancing tolerance.

## VI. RELATED WORK

Different from batch processing and traditional distributed database [7], [19], [27], [28], [33], the problem of load balancing is more challenging on distributed stream processing systems, because of the needs of continuous optimization and difficulty with high dynamism. There are two common classes of strategies to enable load balancing in distributed stream processing systems, namely *operator-based* and *data-based*.

**Operator-based** strategies generally assume the basic computation units are operators. Therefore, load balancing among distributed nodes is achieved by allocating the operators to the nodes. In Borealis [32], for example, the system exploits the correlation and variance of the workloads of the operators, to make more reliable and stable assignments. In [31], Xing et al. observe that operator movement is too expensive for short-term workload bursts. This observation motivates them to design a new load balance model and corresponding algorithms to support more resilient operator placement. System S [29], as another example, also generates scheduling decisions for jobs in submission phase and migrates jobs or sub-jobs to less loaded machines on runtime based on complex statistics, including operators workload and the priority of the applications. Zhou et al. [35] presents a flow-aware load selection strategy to minimize communication cost with their new dynamic assignment strategy adaptive to the evolving stream workloads. In order to improve system balance property, [5] presents a more flexible mechanism by using both online and offline methods under the objective of network traffic minimization. A common problem with operator-based load balancing is the lack of flexible workload partitioning. It could lead to difficulty to any operator-based load balancing technique, when an operator is much more overloaded than all other operators.

**Data-based** strategies allow the system to repartition the workload based on keys of the tuples in the stream, motivated by the huge success of MapReduce system and its variants.

Elastic stream processing is a hot topic in both database and distributed system communities. Such systems attempt to scale out the computation parallelism to address the increasing computation workload, e.g., [12], [30]. By applying queuing theory, it is possible to model the workload and expected



processing latency, which can be used for better resource scheduling [10]. When historical records are available to the system, it is beneficial to generate a long-term workload evolution plan, to schedule the migrations in the future with smaller workload movement overhead [8]. Note that all these systems and algorithms are designed to handle long-term workload variance. All these solutions are generally too expensive if the workload fluctuation is just a short-term phenomenon. The proposal in this work targets to solve the short-term workload variance problem with minimal cost.

A number of research work focus on load balancing in distributed stream join systems. [9] models the join operation on a square matrix, each side of which represents one join stream. It distributes tuples randomly to cells on each line (or column) in matrix, which produces a potential join output. To avoid these problems, in [20], it proposes a join-biclique model which organizes the clusters as a complete bipartite graph for joining big data streams. In [13], it classifies skewness on join key granularities and divides the joins into three types, e.g., B-Skew join, E-Skew join and H-Skew join. It proposes to deal with load imbalance by using different join algorithms. In [25], it designs a two-tiered partitioning method which handles hot tuples separately from cold tuples and distributes keys with heavy keys (big granularities) first. DKG [23] also distinguishes heavy keys from light ones by granularities and applies greedy algorithms for load balance. Although these techniques are effective for stream job applications, they are not directly extensible to general-purpose stream processing.

Flux [24] is the widely adopted load balancing strategy, designed for traditional distributed streaming processing systems. It simply measures the workload of the tasks, and attempts to migrate workload from overloaded nodes to underloaded nodes. One key limitation of Flux is the lack of consideration on the routing overhead. In traditional stream processing systems, the workload of a logical operator is pre-partitioned into tasks, such that each task may handle a huge number of keys but processed by an individual thread at any time. The approach proposed in this paper allows the system to reassign keys in a much more flexible manner. Our approach also takes routing overhead into consideration, because it is unrealistic to fully control the destination for all keys from a large domain.

Nasir et al. [21], [22] design a series of randomized routing algorithms to balance the workload of stream processing operators. Their strategy is based on the theoretical model called power-of-two, which evaluates two randomly chosen candidate destinations for each tuples and chooses the one with smaller workload estimation. Their approach is more appropriate for stateless operators in streaming processing, and a subset of stateful operators by introducing an aggregator to combine results of tuples sent to different working threads. A number of stateful operators, such as join, may need all historical tuples with certain keys in order to generate complete and accurate computation results, which cannot be supported by such scheme. The method proposed in this paper, however, does not have such limitation, thus applicable to any stateful operator with perfect load balancing performance.

Readj [11] is proposed to resolve the stateful load balance problem with a small routing table, which is the most similar work to our proposal. It introduces a similar tuple distribution function, consisting of a basic hash function and an explicit

hash table. However, the workload redistribution mechanism used in Readj is completely different from ours. The algorithm in Readj always tries to move back the keys to their original destination by hash function, followed with migration schedules on keys with relatively larger workload. Their strategy might work well when the workload of the keys are almost uniform. When the workloads of the keys vary dramatically, their approach either fails to find a reasonable load balancing plan, or incurs huge routing overhead by generating a large routing table. The routing algorithms designed in this paper completely tackle this problem, which presents high efficiency as well as good balancing performance in almost all circumstances.

## VII. CONCLUSION AND FUTURE WORK

This paper presents a new dynamic workload distribution mechanism for intra-operator load balancing in distributed stream processing engines. Our mixed distribution strategy is capable of assigning the workload evenly over task workers of an operator, under short-term workload fluctuations. New optimization techniques are introduced to improve the efficiency of the approach, to enable practical implementation over mainstream stream processing engines. Our testings on Apache Storm platform show excellent performance improvement with a variety of workload from real applications, also present huge advantage over existing solutions on both system throughput and response latency. In the future, we will investigate the theoretical properties of the algorithms to better understand the optimality of the approaches under general assumptions. We will also try to design a new mechanism, to support smooth workload redistribution suitable to both long-term workload shifts and short-term workload fluctuations.

## REFERENCES

- [1] Apache Storm. <http://storm.apache.org/>.
- [2] The TPC-H Benchmark. <http://www.tpc.org/tpch>.
- [3] D. Abadi, Y. Ahmad, M. Balazinska, and et al. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
- [4] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, pages 456–467, 2004.
- [5] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *DEBS*, pages 207–218, 2013.
- [6] T. Chen, H. Haussecker, A. Bovyrin, and et al. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9(2), 2005.
- [7] D. Dewitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [8] J. Ding, T. Fu, R. Ma, M. Winslett, Y. Yang, Z. Zhang, and H. Chao. Optimal operator state migration for elastic data stream processing. *Mccarthy*, 2015.
- [9] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *VLDB*, 7(6):441–452, 2014.
- [10] T. Fu, J. Ding, R. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Dynamic resource scheduling for real-time analytics over fast streams. In *ICDCS*, pages 411–420, 2015.
- [11] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDBJ*, 23(4):517–539, 2014.
- [12] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1447–1463, 2014.

- [13] R. Huebsch, M. Garofalakis, J. Hellerstein, and I. Stoica. Advanced join strategies for large-scale distributed computation. *VLDB*, 7(13):1484–1495, 2014.
- [14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.
- [15] N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *Foundations of Computer Science*, pages 312–320, 1982.
- [16] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K. Wu, H. Andrade, and B. Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware*, pages 308–327, 2009.
- [17] S. Kulkarni, N. Bhagat, M. Fu, and et al. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.
- [18] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. Monalytics: Online monitoring and analytics for managing large scale data centers. In *ICAC*, pages 141–150, 2010.
- [19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, pages 25–36, 2012.
- [20] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.
- [21] M. Nasir, G. Morales, D. Garciasoriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pages 137–148, 2015.
- [22] M. Nasir, A. U., G. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. *ICDE*, 2016.
- [23] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *DEBS*, pages 80–91, 2015.
- [24] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.
- [25] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. Elmore, A. Aboulmaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *VLDB*, 8(3):245–256, 2014.
- [26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, and et al. Storm@ twitter. In *SIGMOD*, pages 147–156, 2014.
- [27] N. Ufler, B. and Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *ICDE*, pages 522–533, 2012.
- [28] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, pages 537–548, 1991.
- [29] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K. Wu, and L. Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.
- [30] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734, 2015.
- [31] Y. Xing, J. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.
- [32] Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.
- [33] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, pages 1043–1052, 2008.
- [34] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.
- [35] Y. Zhou, B. Ooi, and K. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, pages 322–323, 2005.

---

## algorithm 5 Simple Algorithm

---

**input:** task instances in  $\mathcal{D}$

**output:**  $A'$

```

1: Disassociate keys from all the instances
2: foreach  $d$  in  $\mathcal{D}$  do
3:    $L(d) \leftarrow 0$ 
4: Add all keys to  $\mathcal{C}$ , i.e.,  $\mathcal{C} \leftarrow \mathcal{K}$ 
5: foreach  $k$  in  $\mathcal{C}$  in descending order of  $c(k)$  do
6:   foreach  $d$  in  $\mathcal{D}$  in ascending order of  $L(d)$  do
7:     Associate key  $k$  with instance  $d$ , i.e.:
8:      $L(d) = L(d) + c(k)$ 
9:     if  $h(k) \neq d$  then
10:      Add entry  $(k, d)$  to  $A'$ 
11:      Remove  $k$  from  $\mathcal{C}$ ;
12: return  $A'$ 

```

---

## APPENDIX

In order to derive theoretic results about the LLFD algorithm, we first look at a more simplified key assignment algorithm, namely the Simple algorithm. We next derive a series of theoretic results based on the Simple algorithm. Lastly, we show how these results are applicable to the LLFD algorithm. As described in Algorithm 5, the *Simple* algorithm works in the following way, at first, it disassociates and puts all the keys into the candidate set  $\mathcal{C}$  (Lines 1–4). Secondly it sorts these keys in a descending order of the computation cost  $c(k)$ . Finally it sequentially assigns each key to the instance with the least total workloads so far (Line 5–11).

*Definition 1:* A perfect assignment is defined as the approach that can assign keys to instances resulting in  $\forall d_i, d_j \in \mathcal{D}, L(d_i) = L(d_j) = \bar{L} = \frac{1}{N_D} \sum_{k \in \mathcal{K}} c(k)$ .

*Lemma 1:* Given the instance set  $\mathcal{D}$  of size  $N_D$ , key set  $\mathcal{K}$  of size  $K$  and computation cost of each key  $c(k)$ , where keys are in a non-increasing order of their computation costs, i.e.,  $c(k_1) \geq c(k_2) \geq \dots \geq c(K)$ , if the perfect assignment exists, we have:

$$c(k_{qN_D+1}) \leq \frac{1}{q+1} \bar{L}, \quad q = 1, 2, \dots, \lfloor \frac{K-1}{N_D} \rfloor. \quad (4)$$

*Proof:* Assuming  $c(k_{qN_D+1}) > \frac{1}{q+1} \bar{L}$ , then we have  $c(k_1) \geq c(k_2) \geq \dots \geq c(k_{qN_D+1}) > \frac{1}{q+1} \bar{L}$ . This means that for keys from  $k_1$  to  $k_{qN_D+1}$ , each instance can at most be associated with  $q$  of them. In result, any instance that is associated with the  $(qN_D+1)$ -th key will generate workloads larger than  $\bar{L}$ , which contradicts the assumption of the existence of the perfect assignment. ■

*Lemma 2:* Given the instance set  $\mathcal{D}$  of size  $N_D$ , key set  $\mathcal{K}$  of size  $K$  and computation cost of each key  $c(k)$ , where keys are in a non-increasing order of their computation costs, i.e.,  $c(k_1) \geq c(k_2) \geq \dots \geq c(K)$ , if the perfect assignment exists and  $c(k_1) < \bar{L}$  (the computation cost of any individual key is smaller than the average workload of task instances), we have  $K \geq 2N_D$ .

*Proof:* This is straight forward given (a) the perfect assignment exists and (b) the computation cost of any individual key is smaller than  $\bar{L}$ , because for each instance, there must be at least two keys assigned to it. ■

*Lemma 3:* Given the instance set  $\mathcal{D}$  of size  $N_D$ , key set  $\mathcal{K}$  of size  $K$  and computation cost of each key  $c(k)$ , where keys are in a non-increasing order of their computation costs, i.e.,  $c(k_1) \geq c(k_2) \geq \dots \geq c(K)$ , if the perfect assignment exists and  $c(k_1) < \bar{L}$ , we have:

$$\theta_{max} \leq \frac{1}{3} \cdot \left(1 - \frac{1}{N_D}\right), \quad (5)$$

where  $\theta_{max} = \max_{d \in \mathcal{D}} \left(\frac{L(d) - \bar{L}}{\bar{L}}\right)$ .

*Proof:* We prove by considering the worst case (in terms of load balance) where (a) the  $(2N_D + 1)$ -th key has the largest possible computation cost  $c(k_{2N_D+1}) = \bar{L}/3$ , according to Lemma 1 and Lemma 2; (b) Keys after the  $(2N_D + 1)$ -th have equal amount of computation costs, denoted by  $\varepsilon$ , which are very close to zero; (c) The remaining workloads, i.e.,  $\sum_{k \in \mathcal{K}} c(k) - \frac{1}{3}\bar{L} - \varepsilon(K - 2N_D - 1)$ , all concentrate on the first  $2N_D$  keys and are evenly distributed, summarized as follows:

$$c(k_i) = \begin{cases} \frac{N_D \bar{L} - \frac{1}{3}\bar{L} - \varepsilon(K - 2N_D - 1)}{N_D} & \text{for } i = 1, 2, \dots, 2N_D; \\ \frac{1}{3}\bar{L} & \text{for } i = 2N_D + 1; \\ \varepsilon & \text{for } i > 2N_D + 1. \end{cases}$$

When  $\varepsilon \rightarrow 0$ , we have:

$$L_{max} = \max_{d \in \mathcal{D}} L(d) = c(k_i) + c(k_{2N_D}) \leq \frac{4}{3}\bar{L} - \frac{\bar{L}}{3N_D},$$

where  $i = 1, 2, \dots, 2N_D$ . Note  $L_{max} = c(k_i) + c(k_{2N_D})$  is because according to the Simple algorithm, keys  $k_i, i > 2N_D + 1$  will never be assigned to the instance with  $L_{max}$ . This completes the proof according to our definition of  $\theta_{max}$ . ■

*Theorem 1:* Given the instance set  $\mathcal{D}$  of size  $N_D$ , key set  $\mathcal{K}$  of size  $K$  and computation cost of each key  $c(k)$ , where keys are in a non-increasing order of their computation costs, i.e.,  $c(k_1) \geq c(k_2) \geq \dots \geq c(K)$ , if the perfect assignment exists and  $c(k_1) < \bar{L}$ , LLFD always finds a solution resulting with balancing indicator  $\theta(d, F)$  no worse than  $\frac{1}{3}(1 - \frac{1}{N_D})$  for any task instance  $d$ .

*Proof:* According to Algorithm 1, it has a larger search space than that of the Simple Algorithm, and is devoted to finding the assignment with more balanced workloads among instances, i.e.,  $\theta(d, F) \leq \theta_{max} \leq \frac{1}{3} \cdot \left(1 - \frac{1}{N_D}\right)$ , which is proved in Lemma 3. ■

*Theorem 4:* Balance status generated by the Mixed represented by  $\theta_{Mix}$  is not worse than the balance status  $\theta_{Sim}$  produced by the Simple algorithm.

*Proof:* Supposing  $\theta_{Mix} > \theta_{Sim}$ , now we take  $\theta_{Sim}$  as  $\theta_{max}$ , then the Mixed algorithm overload instance and  $\exists_{cop}$  in Mixed's migration process. Because the Mixed has tried all instances to put  $c(k_{cop})$  for without overload as shown in Algorithm 4, then  $\forall d, d \in \mathcal{D}$ ,  $c(k_{cop}) + L(d) - \sum_{k' \in \{k'' | c(k'') < c(k_{cop})\}} c(k')$  is larger than upbound. This is contradicts to begin supposing:  $\theta_{Mix} > \theta_{Sim}$ . ■

With the defaulted parameter settings, our *Mixed* algorithm with different numbers of  $N_A$  has different migration cost as shown in Fig. 17. When we set  $N_A$  to the values less than 1000 (calculated by  $2^i$  with  $i \leq 10$ ) for  $\theta_{max} = 0.08$ , our algorithm generates a large migration cost for it acts as algorithm (Alg. 2)

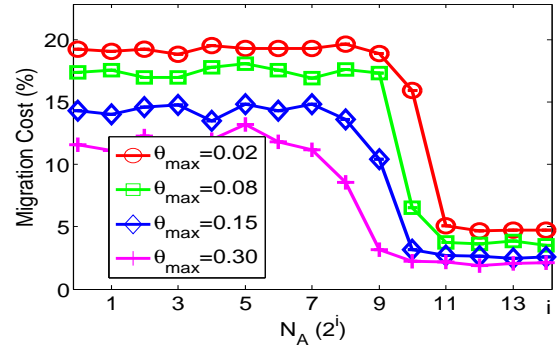


Fig. 17. Migration cost with different  $N_A$  by *Mixed*.

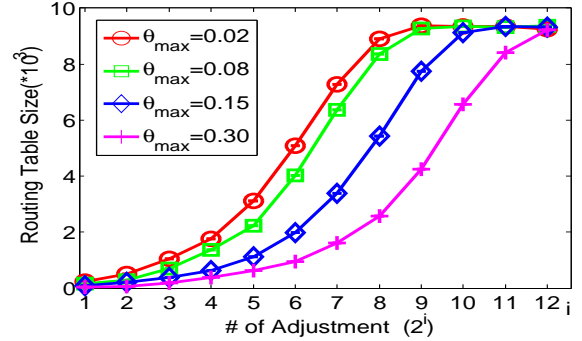


Fig. 18. Routing table changing along with # of adjustments.

and leaves alone  $N_A$  requirement. But when  $N_A$  is relaxed to 2000 ( $i > 11$ ), migration cost decreases greatly, since our algorithm acts according to *Mixed* algorithm which can reduce the routing back actions greatly. Furthermore, the different degrees of balance status ( $\theta_{max}$ ) require different minimum  $N_A$ , and it will drastically reduce the migration cost shown in Fig. 17.

Fig. 18 shows the change of routing table size for different numbers of adjustment. We use algorithm (Alg. 3) to do balance and set  $K = 10^4$  to verify the results quickly. Obviously, the smaller  $\theta_{max}$  accelerates the growth of routing table. We also observe that routing table sizes with different  $\theta_{max}$ s converge to the same size (around 9350 entries) in Fig. 18. This is because *Min Mig* does balancing without considering the constraint for routing table size. In other words, a key is paired to the task randomly when the basic assignment function has caused imbalance. Therefore, the probability of a key appears in routing table is  $\frac{N_D - 1}{N_D}$ , and then, after a long period of load balancing, the routing table size should be  $\frac{N_D - 1}{N_D} \cdot K$  with  $K$  as the key size.

We show the migration cost with different window size in Fig. 19. Since the larger window size provides more chance for finding the appropriate migration keys ( $\gamma_i(k, w)$ ), the migration cost of *Mixed* is smaller than *MinTable*.

To characterize both computation and migration cost, we propose the *migration priority index* for each key, defined as  $\gamma_i(k, w) = c_i(k)^\beta S_i(k, w)^{-1}$  as in Sec. III-B.  $\beta$  is used to measure the importance of computation cost and the memory consumption which decides the *migration priority* of keys. In Alg. 3, a key with the larger  $\gamma_i(k, w)$  has the higher priority

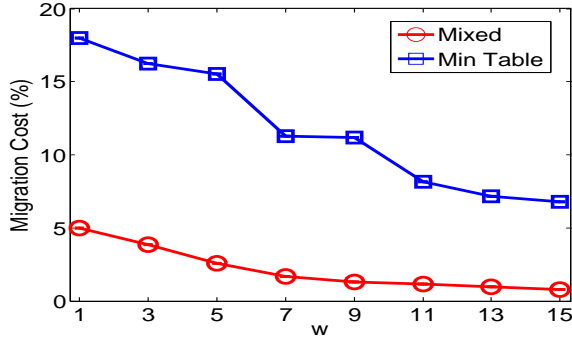


Fig. 19. Migration cost vs window size.

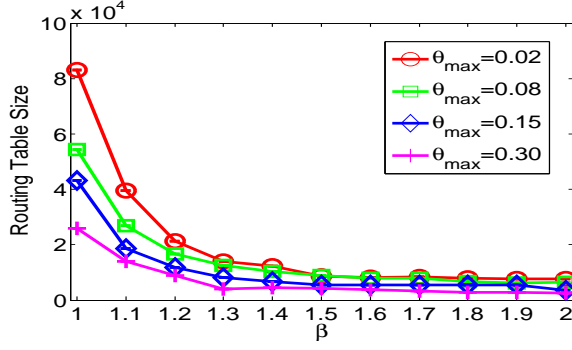


Fig. 20. Routing table size in different  $\beta$ .

to be migrated. Larger  $\beta$  represents that the migration method concerns more on faster computation rather than on less migration cost (memory consumption). Furthermore, larger  $\beta$  will produce smaller routing table since the migration method preferentially migrates keys with large load. Fig. 20 and Fig. 21 show the change of routing table size and migration cost with different values of  $\beta$ . Those results are produced by the *Min-Mig* algorithm, and each result is an average value generated by running 10 times of balance adjustments. In Fig. 20,  $\beta = 1$  means the migration candidates are evaluated according to the load per unit memory consumption. In this case, keys with smaller load would be selected and a larger routing table will be generated. As  $\beta$  is set larger, migration method gradually tends to move the bigger load keys, then routing table size becomes smaller. Furthermore, when  $\beta \in [1.5, 2]$ , routing table size is stable because the migration candidates are almost

selected only by the load of keys. The lines in Fig. 21 show results by the influence of parameter  $\beta$ . Based on these sets of parameter tests, we select  $\beta = 1.5$  as the default value in our experiments.

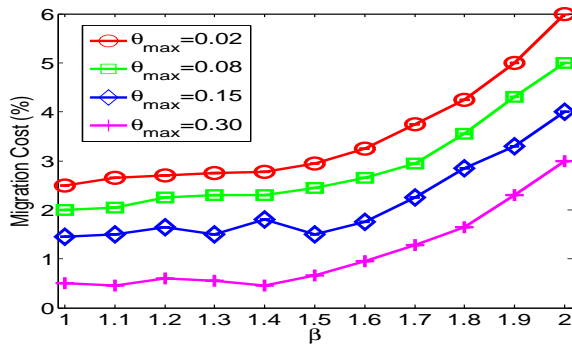


Fig. 21. Migration cost in different  $\beta$ .