Reshape: Adaptive Result-aware Skew Handling for Exploratory Analysis on Big Data

Avinash Kumar, Sadeem Alsudais, Shengquan Ni, Zuozhi Wang, Yicong Huang, Chen Li
Department of Computer Science, UC Irvine, CA 92697, USA
{avinask1,salsudai,shengqun,zuozhiw,yicongh1,chenli}@ics.uci.edu

ABSTRACT
The process of data analysis, especially in GUI-based analytics systems, is highly exploratory. The user iteratively refines a workflow multiple times before arriving at the final workflow. In such an exploratory setting, it is valuable to the user if the initial results of the workflow are representative of the final answers so that the user can refine the workflow without waiting for the completion of its execution. Partitioning skew may lead to the production of misleading initial results during the execution. In this paper, we explore skew and its mitigation strategies from the perspective of the results shown to the user. We present a novel framework called Reshape that can adaptively handle partitioning skew in pipelined execution. Reshape employs a two-phase approach that transfers load in a fine-tuned manner to mitigate skew iteratively during execution, thus enabling it to handle changes in input-data distribution. Reshape has the ability to adaptively adjust skew-handling parameters, which reduces the technical burden on the users. Reshape supports a variety of operators such as HashJoin, Group-by, and Sort. We implemented Reshape on top of two big data engines, namely Amber and Flink, to demonstrate its generality and efficiency, and report an experimental evaluation using real and synthetic datasets.

CCS CONCEPTS
• Computing methodologies → Distributed algorithms.

KEYWORDS
distributed system, partitioning skew, data processing workflow

1 INTRODUCTION
As information volumes in many applications become large, data analytics is becoming increasingly important. Data processing frameworks such as Hadoop [5], Spark [7], and Flink [4] provide programming interfaces that are used by developers to code their data processing needs. GUI-based workflow systems such as Alteryx [3], RapidMiner [46], Knime [34], Einblick [24], and Texera [55] provide a GUI interface where the users can drag-and-drop operators and create a workflow as a directed acyclic graph (DAG). Once the data processing job is created, it is submitted to an engine that executes the job.

The process of data analysis, especially in GUI-based analytics systems, has two important characteristics. 1) Highly exploratory: The process of building a workflow can be very exploratory and iterative [27, 53, 57]. Often the user constructs an initial workflow and executes it to observe a few results. If they are not desirable, she terminates the current execution and revises the workflow. The user iteratively refines the workflow until finishing a final workflow to compute the results. As an example, Figure 1 shows a workflow at an intermediate step during the task of covid data analysis. It examines the relationship between the number of tweets containing the keyword covid and the number of Covid cases in 2020. The monthly details about the Covid cases are joined with tweets filtered on the covid keyword on the month column. The result is plotted as a visualization operator that shows a bar chart about the total count of tweets about Covid per month and a line chart about the total Covid cases per month. The analyst may observe the visualization and choose to continue refining the workflow to do analysis for specific US states. In the Texera system we are developing, we observe that the users refined a workflow about 80 times on an average before reaching the final version. 2) Suitable for non-technical users: GUI-based workflow execution systems significantly lower the technical learning curve for its users, thus enabling non-IT domain experts to do data science projects without writing code. Such systems also try to minimize the requirements on users to know the technical details of workflow execution, so that the user can focus solely on the analytics task at hand.

In exploratory data analytics, it is vital for a user to see results quickly to allow her to identify problems in the analysis early and take corrective actions without waiting for the entire workflow to finish executing. Pipelined execution [14] is a popular workflow execution model that can produce results quickly. In pipelined execution, an operator does not wait for its entire input data to be produced before processing the input and sending results to its downstream operators. For example, when the workflow in Figure 1 is executed using pipelined model, the HashJoin operator starts executing and producing results as soon as the Filter operator outputs initial results. The user can notice the initial results and...
make any changes, if needed. Pipelined execution is adopted by data-processing engines such as Flink [4], Samza [6], and Storm [8].

As data volumes in these systems increase, it is indispensable to do parallel processing, in which data is partitioned and processed by multiple computation units in parallel. Data partitioning, either using hash partitioning or range partitioning, often results in skew. As an example, the HashJoin operator in Figure 1 receives hash partitioned inputs from the two upstream operators. Although the hash function allots the same number of months to each join worker, load imbalance still exists because of different numbers of tweets for those months. It is well known that partitioning skew adversely affects the efficiency of engines as it increases the processing time and reduces the throughput [20, 37].

The problem of partitioning skew has been extensively studied in the literature, mainly from the perspective of increasing the end-to-end performance. However, there is little research on the following important problem:

**In exploratory data analytics, how to consider the results shown to the user when mitigating skew?**

In exploratory data analysis, it is valuable to the analyst if the initial results are representative of the final results because they allow her to identify issues early and make necessary changes. Partitioning skew may lead to the production of misleading results during the execution. Let us consider the production rate of October and December tuples from the HashJoin operator in the running example. Assume that the HashJoin operator is the bottleneck of the execution, and its workers receive input at an equal or higher rate than they can process. Although there are more December tuples than October, their production rates are similar because the total amounts of data received by $J_4$ and $J_6$ are different (details in Section 3). Thus, the bar chart shows similar heights for October and December bars till $J_4$ completes processing, whereas the December bar is almost four times taller than the October bar in the final result. In this paper, we explore partitioning skew mitigation in the setting of exploratory data analysis and analyze the effect of mitigation strategies on the results shown to the user.

A common solution to handle partitioning skew at an operator is blocking the partitioning of its input data till the entire input data is produced by its upstream operator [2, 19, 20, 50, 54] and then sampling the input data to create an optimal partitioning function. For example, in Figure 1, the HashJoin operator waits for the Filter operator to completely finish. Then, the output of the Filter operator is sampled to create an optimal partitioning function to send data to the HashJoin operator. Such blocking is not allowed in pipelined execution which makes these solutions infeasible in pipelined execution setting. Even temporally blocking the partitioning till a small percentage of input (e.g., 1% [47]) is collected for sampling can result in a long delay if there is an expensive upstream operator.

A different solution applicable to the pipelined execution setting is to detect the overloaded workers of an operator at runtime and transfer the processing of a few keys of the overloaded worker to a more available worker. For example, $J_6$ is detected to be overloaded at runtime and the processing of June tuples is transferred to $J_4$. However, this transfer has little effect on the results shown to the user (details in Section 3). In order to show representative initial results, the data of December has to be split between $J_4$ and $J_6$. Thus, these two approaches of transferring load from $J_6$ to $J_4$ have different impacts on the initial results shown to the user.

In this paper, we analyze the effect of different skew mitigation strategies on the results shown to the user and present a novel skew handling framework called Reshape that adaptively handles skew in a pipelined execution setting. Reshape monitors the workload metrics of the workers and adapts the partitioning logic to transfer load whenever it observes a skew. These modifications can be done multiple times as the input distribution changes [13, 35] or if earlier modifications did not fully mitigate the skew. The command to adapt the partitioning logic is sent from the controller to the workers using low latency control messages that are supported in various engines such as Flink, Chi [38], and Amber [36].

We make the following contributions. (1) Analysis of the impact of mitigation on the shown results: We present different approaches of skew mitigation and analyze their impact on the results shown to the user (Section 3). (2) Automatic adjustment of the skew detection threshold: We present a way to dynamically adjust the skew detection threshold to reduce the number of iterations of mitigation to minimize the technical burden on the user (Section 4). (3) Applicability to multiple operators: Since a data analysis workflow can contain many operators that are susceptible to partitioning skew, we generalize Reshape to multiple operators such as HashJoin, Group-by, and Sort, and discuss challenges related to state migration (Section 5). (4) Generalization to broader settings: We consider settings such as high state-migration time and multiple helper workers for an overloaded worker and discuss how Reshape can be extended in these settings (Section 6). (5) Experimental evaluation: We present the implementation of Reshape on top of two big-data engines, namely Amber and Flink, to show the generality of this approach. We report an experimental evaluation using real and synthetic datasets on large computing clusters (Section 7).
1.1 Related work

There have been extensive studies about skew handling in two major execution paradigms in big data engines – batch execution and pipelined execution. Batch-execution systems such as MapReduce [5] and Spark [7] materialize complete input data before partitioning it across workers. Pipelined-execution systems such as Flink [4], Storm [8] and Amber [36] send input tuples to a receiving worker immediately after they are available. The complete input is not known to the operator in pipelined execution, which makes skew handling more challenging.

**Skew handling in batch execution.** A static technique is to sample and obtain the distribution of complete input data and use it to partition data in a way that avoids skew [2, 19, 20, 50, 54]. Adaptive skew-handling techniques adapt their decisions to changing runtime conditions and mitigate skew in multiple iterations. For instance, SkewTune [37] and Hurricane [15] handle skew adaptively. That is, upon detecting skew, SkewTune stops the executing workers, re-partitions the materialized input, and starts new workers to process the partitions. Hurricane clones overburdened workers and uses a special storage that allows fine-grained data access to the original and cloned workers in parallel. Hurricane can split the processing of a key over multiple workers and thus has a fine load-transfer granularity. SkewTune cannot split the processing of a key.

**Static skew handling in pipelined execution.** Flow-Join [47] avoids skew in a HashJoin operator. It samples the first 1% of input data of the operator to decide the overloaded keys and does a broadcast join for the overloaded keys. Since it makes the decision based on an initial portion of the input, it cannot handle skew if the input distribution changes multiple times during the execution. Partial key grouping (PKG) [41, 42] uses multiple pre-defined partitioning functions. It results in multiple candidate workers sharing the processing of the same key. Since the partitioning logic is static, a worker may process multiple skewed keys, which makes it more burdened than other workers. PKG cannot be used to handle skew in operators such as Sort and Median.

**Adaptive skew handling in pipelined execution.** Flux [48] divides the input into many pre-defined mini-partitions that can be transferred between workers to mitigate skew. Thus, the load-transfer granularity is fixed and pre-determined. Also, it cannot split the load of a single overloaded key to multiple workers. Another adaptive technique minimizes the input load on workers that compute theta joins by dynamically changing the replication factor for data partitions [25]. This approach uses random partitioning schemes such as round-robin and hence is not prone to partitioning skew. Reshape handles skew adaptively over multiple iterations. It determines the keys to be transferred dynamically and allows an overloaded key to be split over multiple workers for mitigation.

2 RESHAPE: OVERVIEW

We use Figure 2 to give an overview of Reshape.

2.1 Skew detection

During the execution of an operator, the controller periodically collects workload metrics from the workers of the operator to detect skew (Figure 2(a)). There are different metrics that can represent the workload on a worker such as CPU usage, memory usage and unprocessed data queue size [15, 26]. Skew handling in Reshape is independent of the choice of workload metric, and we choose unprocessed queue size as a metric in this paper. We choose this metric because the results seen by the analyst depend on the future results produced by a worker, which in turn depend on the content of its unprocessed data queue. We refer to a computationally overburdened worker as a skewed worker and workers that share the load as its helper workers.

**Skew test.** Given two workers of the same operator, say C and L, the controller performs a skew test to determine whether C is a helper candidate for L. The skew test uses the following inequalities to check if L is computationally burdened and the workload gap between L and C is big enough:

\[ \phi_L \geq \eta, \]  
\[ \phi_L - \phi_C \geq \tau, \]  

where \( \eta \) and \( \tau \) are threshold parameters and \( \phi_w \) is the workload on a worker w.

**Helper workers selection.** The skew tests may yield multiple helper candidates for L. For simplicity, we assume till Section 5 that one helper worker is assigned per skewed worker. In Section 6, we generalize the discussions by considering multiple helpers per skewed worker. The controller chooses the helper candidate with the lowest workload that has not been assigned to any other overloaded worker as the helper of L. In our discussions forward, we use S and H to refer to a skewed worker and its chosen helper worker respectively.

2.2 Skew mitigation

Suppose the skewed worker S and its helper H have been handling input partitions \( I_S \) and \( I_H \), respectively (Figure 2(a)). Reshape transfers a fraction of the future input of S to H to reduce the load on S. Here future input refers to the data input that is supposed to be received by a worker but has not yet been sent by the previous operator. The controller notifies S about the part \( p \) of partition \( I_S \) that will be shared with H to reduce the load of S (Figure 2(b)). The downstream results shown to the user have a role to play in deciding \( p \), which will be discussed in Section 3. Worker S sends to H its state.
information $State_p$ corresponding to the partition $p$ (Figure 2(c)). Details about state-migration strategies are in Section 5. We assume the state migration time to be small till Section 5. In Section 6, we consider the general case where the state-migration time can be significant. Worker $H$ saves the state information and sends an $ack$ message to $S$, which then notifies the controller (Figure 2(d)). The controller changes the partitioning logic at the previous operator (Figure 2(e,f)).

**Fault Tolerance.** The Reshape framework supports the fault tolerance mechanism of the Flink engine [17] that checkpoints the states of the workers periodically. During checkpointing, a checkpoint marker is propagated downstream from the source operators. When an operator receives the marker from all its upstream operators, it takes a checkpoint which saves the current states of the workers of the operator. Every checkpoint has the information about the current partitioning logic at the workers. If checkpointing occurs during state migration, then the skewed worker additionally forwards the checkpoint marker to each of its helper workers. A helper worker needs to wait for the checkpoint marker from its corresponding skewed worker. Since the skewed workers and the helper workers are two disjoint sets of workers, there is no cyclic dependency in marker propagation and the checkpointing process successfully terminates. During recovery, the workers restore their states from the most recent checkpoint and then continue the execution.

## 3 RESULT-AWARE LOAD TRANSFER

After helper workers are selected for the skewed workers, the load needs to be transferred from the skewed workers to the corresponding helper workers. In Section 3.1, we consider the different approaches of load transfer between workers and analyze their impact on the results shown to the user. Unlike other skewed handling approaches that focus on evenly dividing the future incoming load among the workers, Reshape has an extra phase of load transfer at the beginning that removes the existing load imbalance between the workers. In Section 3.2, we discuss these two phases of load transfer and the significance of the first phase.

### 3.1 Mitigation impact on user results

There are broadly two approaches to transfer the load from a skewed worker to its helper worker. We use the probe input of the HashJoin operator in Figure 1 (from the Filter operator) as an example to explain the concepts in this section. It is assumed that the build phase of the join has finished. Suppose Reshape detects $J_6$ and $J_5$ as the skewed workers in the running example and $J_4$ and $J_2$ are their corresponding helpers, respectively. The load-transfer approaches are implemented by changing the partitioning logic at the Filter operator and affects the future tuples going into the HashJoin operator.

2. **Split by records (SBR).** In this approach, the records of the keys in the partition of the skewed worker are split between the skewed and the helper worker. The ratio of the split decides the amount of load transferred to the helper worker. For example, if the Filter operator needs to redirect $\frac{7}{9}$ of the input $J_6$ to $J_4$, then it redirects 9 tuples out of every 26 tuples in $J_6$’s partition to $J_4$.

**Impact of the two approaches on user results.** The two load transfer approaches have their own advantages and limitations. For example, SBR incurs an extra overhead compared to SBR because SBR requires the workers to store the distribution of workload per key. On the other hand, SBR may require transfer of a larger state size compared to SBR, if all the keys of a skewed worker are shared with the helper. There are existing works in literature that address these concerns [29, 31, 39, 40, 47, 58]. In the remainder of this subsection, we compare these two approaches from the perspective of their effects on the results shown to the user.

#### a) Representative initial results

As discussed before, it is valuable to the user if the initial results are representative of the final results. Partitioning skew may lead to the production of misleading results during the execution as shown next. Let us consider the bar chart visualization for October and December in the running example. The total count of December tweets, according to Figure 1(c), is about four times that of October tweets, i.e., the December bar is about four times longer than the October bar in the final visualization. Assume that the join operator is the bottleneck of the execution, and its workers receive input at an equal or higher rate than what they can process. Also assume that the processing speeds of the workers of HashJoin are the same, say $t$ per second. $J_4$ produces $\frac{7}{9} + t$ October tuples and $J_6$ produces $\frac{2}{9} + t$ December tuples per second in the unmitigated case (Figure 3(a)). The rate of production of October and December tuples are similar because the total amount of data received by $J_4$ and $J_6$ are different. The bar chart shows similar heights for October and December bars in the unmitigated case till $J_4$ completes its processing.

![Figure 3: SBR splits December tuples on both workers and shows representative bar charts.](image)

When SBR is used to mitigate the skew, the processing of June tuples is transferred to $J_4$ (Figure 3(b)). However, this transfer has little effect on the results shown to the user. The production rates of October and December after the transfer are $\frac{7}{9} + t$ and $t$ respectively. That is, the heights of the December and October bars are still about the same, which is not representative of the final results.

SBR has more flexibility for transferring load than SBR because SBR can split the tuples of a key over multiple workers. It leads to more representative initial results than SBR as shown next. The processing of December and June tuples can be split between $J_6$ and $J_4$. For simplicity of calculation, we assume that only December tuples are shared with $J_6$. Since December tuples are now processed by two workers instead of one, the speed of production of these
tuples increases. In order to make the future workloads of \( J_4 \) and \( J_6 \) similar, SBR redirects \( \frac{20}{9} \) of the input of \( J_6 \) to \( J_4 \), which increases the total percentage load on \( J_4 \) to 16 and decreases that on \( J_6 \) to 17. This is implemented by redirecting 9 December tuples out of every 26 tuples in \( J_6 \)’s partition to \( J_4 \). The production rates of October tuples after the transfer is \( \frac{20}{9} + t \). The December tuples are produced by \( J_4 \) and \( J_6 \). The production rate of December by \( J_4 \) is \( \frac{9}{16} + t \) and by \( J_6 \) is \( \frac{16}{17} + t \), which results in a total of approximately \( \frac{24}{17} + t \). Thus, using SBR leads to a more representative production ratio of December to October tuples of about 24:6, which is similar to the actual ratio of 25:6.

![Figure 4: Processing a key at multiple workers by SBR leads to a broken line chart. Only December tuples have been shown for simplicity.](image)

b) Preserving order of tuples. If the tuples of a key being input into an operator are in a particular order and they need to be processed in that order, then SBK is the suitable approach because it enforces a processing order by restricting the processing of the tuples of a key to a single worker at a time. If the processing of a key needs to be transferred to another worker, the migration can be synchronized using techniques such as pause and resume [9, 16, 48] or markers [25] (details in Section 5) so that the tuples are processed in order. In contrast, SBR distributes the tuples of a key over multiple workers to be processed simultaneously, which may cause them to be processed out of order. Consider the following example where an out-of-order processing of the tuples of a key is not desirable. Let us slightly modify the visualization operator in the running example to plot a line chart that shows daily count of covid related tweets. The daily count for each month is plotted as a separate line in the line chart. Figure 4 shows the plot for December in the line chart. Applications may want to show such plots as a continuous line with no breaks, starting from day 1 and extending towards increasing dates as execution progresses, for user experience purposes [1]. In order to achieve this, the tuples of a monthly input into the HashJoin operator are sorted in the increasing order of date. It is expected that HashJoin produces tuples sorted by date, which can be consumed by the visualization operator to create a continuous plot.

SBK assures that the December key is processed by only one worker at a time. Thus, it preserves the order of December tuples in the output sent to the visualization operator (Figure 4(a)). When SBR is used, the December tuples are split between \( J_4 \) and \( J_2 \). In the example shown in Figure 4(b), the Filter operator starts partitioning December tuples by SBR when the tuples around the 26th of December are being produced by the Filter operator. Consequently, \( J_2 \) starts receiving the tuples from the date of the 26th of December and above. As \( J_2 \) and \( J_4 \) concurrently process data, the visualization operator receives the tuples out of order, resulting in broken line chart plots as shown in the figure.

In conclusion, SBR allows more flexibility and enables the production of representative initial results than SBK, but SBR does not preserve the order of tuples. Thus, SBR can be chosen unless there exists a downstream operator that imposes some requirement over the input order of the tuples. Such operators can be found at the workflow compilation stage. The operators before such an operator in the workflow can adopt SBK.

### 3.2 Extra phase in load transfer

The goal of skew mitigation is to use one of the two approaches to transfer the load from the skewed worker to the helper worker in such a way that both workers have a similar workload for the rest of the execution. The skew handling works in literature usually have a single phase of load transfer that focuses on splitting the incoming input such that the workers receive similar load in future. Reshape has an extra phase of load transfer at the beginning that removes the existing load imbalance between the workers. We first give an overview of the two phases in Reshape, and explain the significance of the first phase.

![Figure 5: An implementation of the two phases using the “SBK” and “SBR” approach. \( X_1 \) is a previous operator worker.](image)

**First Phase.** After the detection of skew (Figure 5(a)), the controller issues the first phase of load transfer. The first phase lets the helper “catch up” quickly with the skewed worker. One implementation of the partitioning logic in the first phase at the Filter operator is that it sends all future tuples of \( J_6 \) to \( J_4 \) (Figure 5(b)). Note that \( J_6 \) will continue to process the data in its queue. An alternative implementation is to send only a portion of \( J_6 \)’s partition, such as the December data, to \( J_4 \). This alternative reduces the amount of state transfer, but it will take longer time for \( J_4 \) to catch up with \( J_6 \).

**Second Phase.** Once the queue sizes of the two workers become comparable, the controller starts the second phase. Its goal is to modify the partitioning logic at the Filter operator to redirect part of the future input of \( J_6 \) in such a way that both the workers receive a comparable workload. In order to do this, first the incoming workload of the workers needs to be estimated. A sample of workloads needs to be collected to estimate the future workload of the workers [19, 29, 45, 58] using a prediction function \( \psi \). Reshape can use the sample from the recent history collected during the current execution [33, 49]. If historical data is available, it can complement the recent data and improve the prediction accuracy [28, 43].

To simplify the discussion of the second phase, we make the following assumptions:

- The two workers receive data at constant rates.
- We have a perfect estimator to accurately predict the incoming data workload on the workers.
In Section 4 we will relax these two assumptions. In Figure 1(c), the original load ratio of $J_0$ to $J_4$ is 26:7. SBK cannot handle the skew between $J_0$ and $J_4$. The approach transfers the June month to $J_4$ (Figure 5(c1)), which does not mitigate the skew. However, SBR can redirect $\frac{5}{7}$ of the input of $J_0$ to $J_4$, which mitigates the skew by increasing the percentage load on $J_4$ to 16 and decreasing the percentage load on $J_0$ to 17. An example where SBK can mitigate the skew is the case of skew between the skewed worker $J_0$ and its helper $J_2$. SBK can transfer the processing of May to $J_2$, which brings the two workers to a similar workload. Specifically, the percentage load on $J_2$ increases to 18 and that on $J_0$ decreases to 15.

It should be noted that two phases do not mean that the state transfer has to be done twice necessarily. There are implementations where the state transfer during the first phase is enough and the second phase does not require another state transfer. For example, in SBR, the state of all keys are sent to $J_0$ in the first phase, and there is no state migration needed for the second phase.

**Significance of the first phase.** Reshape has an extra phase for two reasons. First, it gives some immediate respite to the skewed worker and avoids imminent risks of the skewed worker going out of computing resources, invoking back-pressure [11] etc. Second, it may allow the user to see the representative results earlier compared to the case where there is only one phase. Figure 6 illustrates this idea. For simplicity of calculation, we assume that $J_4$ processes October and $J_0$ processes December only. Notice that December tuples are almost four times the tuples of October (Figure 1(c)). Suppose the Hashjoin operator receives 2 October and 8 December tuples every second and the skew is detected when the unprocessed queue sizes of $J_4$ and $J_0$ are 10 and 40, respectively. Figure 6(a) shows the case where there exists a first phase. Suppose the first phase redirects all December tuples to $J_4$. In 3 seconds, $J_4$ receives 24 December and 6 October tuples and catches up with the queue of $J_0$. After this, the second phase starts and redirects 3 out of every 8 December tuples to $J_0$. Assuming the workers process tuples at similar rates, the bar charts show the October and December tuples count shown to the user as the workers process more data. When the workers have processed 10 tuples each, the bar chart shows 10 tuples for both months. After that the effect of first phase starts. When both workers have processed 40 tuples each, the bar chart shows 16 tuples for October and 64 tuples for December, which is representative of the ratio of October to December tuples in the input data. Figure 6(b) shows the case where there is no first phase. After detection of skew, the second phase starts and redirects 3 out of every 8 December tuples to $J_0$. In this case, even after both the workers have processed 40 tuples each, the bar chart shows 22 tuples for October and 58 tuples for December. The ratio gradually moves towards the actual ratio of 1:4 between October to December tuples.

### 4 Adaptive Skew Handling

In the previous section, we assumed that data arrives at constant rates to the workers and the second phase has a perfect estimator. In this section, we study the case when these assumptions are not true. In particular, variable patterns in incoming data rates and an imperfect estimator can result in erroneous workload predictions. Consequently, the second phase may not be able to keep the workload of the skewed and helper workers at a similar level. Thus, the controller may start another iteration of mitigation. Since, each iteration may incur an overhead, such as state transfer, we should try to make better workload predictions so that the number of iterations is reduced. We show that the workflow prediction accuracy depends on the skew detection threshold $\tau$ (Sections 4.1 and 4.2). In order to reduce the technical burden on the user to fix an appropriate $\tau$, we develop a method to adaptively adjust $\tau$ to make better workload predictions (Section 4.3).

#### 4.1 Load reduction from mitigation

We measure the load reduction (LR) from mitigation as the difference in the maximum input size received by a skewed worker and its helper without and with mitigation. Formally, let $S$ and $H$ represent the skewed worker and the helper worker, respectively. The load reduction is defined as:

$$LR = \left[\max(\sigma_S, \sigma_H)\right]_{\text{unmitigated}} - \left[\max(\sigma_S, \sigma_H)\right]_{\text{mitigated}}. \quad (3)$$

where $\sigma_w$ is the size of the total input received by a worker $w$ during the entire execution.

In Figure 7, $D$ represents the difference in the total input sizes of $S$ and $H$ in the unmitigated case. When mitigation is done, due to workload estimation errors, the second phase may not be able to redirect the precise amount of data to keep the workloads of $S$ and $H$ at a similar level. In Figure 7(a), less than $D \frac{7}{8}$ tuples of $S$ are redirected to $H$. Thus, $S$ receives more total input than $H$ and the load reduction is less than $\frac{D}{8}$. Similarly, in Figure 7(b), more than $D \frac{3}{8}$ tuples of $S$ are redirected to $H$. As a result, the load reduction is again less than $\frac{D}{8}$. The ideal mitigation, shown in Figure 7(c), makes the total input of the two workers equal so that they finish around the same time. In particular, $D \frac{3}{8}$ tuples of $S$ are sent to $H$, which is the maximum load reduction ($LR_{\text{max}}$) that can be achieved.

**Figure 7**: Effect of the amount of transferred data on the load reduction (shown in blue). The colored boxes represent the input of $S$ redirected to $H$ in the two phases.

#### 4.2 Impact of $\tau$ on load reduction

In this subsection, we discuss how the load reduction is affected by the value of $\tau$ at which the mitigation starts. Assume that the
operator can have only one iteration of mitigation consisting of two phases. If the second phase uses a perfect estimator and the incoming data rates are constant, as assumed in Section 3, then the maximum load reduction of $D/2$ can be achieved. That is:

$$LR_1 + LR_2 = D/2,$$

(4)

where $LR_1$ and $LR_2$ are the load reduction resulting from the first phase and second phase, respectively.

In general, the workloads estimations have errors [18, 19, 29, 45, 58]. These errors can cause the second phase to redirect less or more than the ideal amount of $S$ tuples (Figure 7(a,b)). In other words, the load reduction from the second phase depends on the accuracy of workload estimation. The workload estimation accuracy depends on $\tau$ as shown next. If $\tau$ increases, then it takes a longer time for the workload difference of $S$ and $H$ to reach $\tau$, resulting in a higher sample size. Suppose the estimation accuracy increases as the sample size increases. Then a higher $\tau$ means that the system makes a more accurate workload estimation. Thus, the total load reduction can be computed as the following:

$$LR = LR_1 + (1 - f(\tau))LR_2,$$

(5)

where $f(\tau)$ is a function representing the error in the estimation of the future workloads. As $\tau$ increases, $f(\tau)$ decreases.

The above analysis shows that a higher $\tau$ results in a higher load reduction. However, setting $\tau$ to an arbitrarily high value means that the system waits a long time before starting the mitigation. Consequently, there may not be enough future input left to mitigate the skew completely. Thus the value of $\tau$ should be chosen properly to achieve a balance between a high estimation accuracy and waiting so long that the opportunity to mitigate skew is lost. This is a classic exploration-exploitation dilemma [10].

Section 4.3.1 discusses how multiple iterations of mitigation are performed. In the previous subsection, we saw that $\tau$ should be chosen appropriately to maintain a balance between workload estimation accuracy and a long delay in the start of mitigation. Section 4.3.2 shows how to autotune $\tau$ adaptively to make better workload estimations, rather than asking the user to supply an appropriate value of $\tau$.

### 4.3.1 Multiple iterations of mitigation

Figure 9 shows an example timeline of two successive iterations of mitigation. The first iteration starts at $t_1$ when the difference of the workloads of $S$ and $H$ exceeds $\tau$. Their workloads are brought to a similar level at $t_2$. Then, the second phase starts. Due to workload estimation errors, the second phase redirects less than the ideal amount of tuples. Thus, the workload of $S$ gradually becomes greater than $H$. At $t_3$, their workload difference exceeds $\tau$ and the second iteration starts.

![Figure 9: Multiple mitigation iterations](image)

### 4.3.2 Dynamically adjusting $\tau$

A low value of $\tau$ causes high errors in workload estimation due to a small sample size, which in turn results in more mitigation iterations. On the other hand, a high $\tau$ may start the mitigation too late when there are not enough future tuples to mitigate the skew. Rather than using a fixed user-provided value of $\tau$, which may be too low or too high, we adaptively adjust $\tau$’s value during execution to make better workload predictions, reduce the number of iterations, and achieve higher load reduction.

In Section 3, we introduced an estimation function $\psi$ that uses a workload sample to estimate future workloads. Let $\epsilon$ denote the standard error of estimation [44], which is a measure of predicted error in workload estimation. For example, the standard error for mean-model [44, 51] estimator is $\epsilon = d\sqrt{1 + \frac{1}{n}}$, where $d$ is the sample standard deviation and $n$ is the sample size. As mentioned in Section 4.2, $\epsilon$ decreases as $\tau$ increases. We want $\epsilon$ to be in a user-defined range $[\epsilon_L, \epsilon_U]$, where $\epsilon_L$ and $\epsilon_U$ are the lower and upper limits, respectively. In particular, when $\epsilon > \epsilon_U$, we assume the error is too high and will lead to a low load reduction. Similarly, when $\epsilon < \epsilon_L$, the error is low enough to make a good estimation.

The controller keeps track of $\epsilon$ and adaptively adjusts $\tau$ in order to move $\epsilon$ towards the $[\epsilon_L, \epsilon_U]$ range. Algorithm 1 describes the process of adjusting $\tau$. For a worker $w$, let $\delta_w^H$ represent the current workload and $\phi_w^H$ represent the workload predicted by $\psi$. The controller periodically collects the current workload metrics from the workers (line 1) and adds them to the existing sample (line 2). The function $\psi$ uses the workload sample to predict future workloads.
Till now we used the running example of skew in the probe input. To return, 10 operator as keyed states and in this subsection, we define two types of operator states, namely immutable states and mutable states. When an operator receives input partitioned by keys, the state information of keys is stored in the operator as keyed states [16]. Each keyed state is a mapping of type $\text{scope} \rightarrow \text{val}$, where $\text{scope}$ is a single key or a set or range of keys, and $\text{val}$ is information associated with the $\text{scope}$. For example, in HashJoin, each join key is a $\text{scope}$, and the list of build tuples with the key is the corresponding $\text{val}$. Similarly, in a hash-based implementation of group-by, each individual group is a $\text{scope}$, and the aggregated value for the group is the corresponding $\text{val}$. In a range-partitioned sort operator, a range of keys is a $\text{scope}$, and the sorted list of tuples in the range is the corresponding $\text{val}$. In the rest of this section, for simplicity, we use the term “state” to refer to “keyed state.”

An input tuple uses the state associated with the $\text{scope}$ of the key of the tuple. If the $\text{val}$ of this $\text{scope}$ cannot change, we say the state is immutable; otherwise, it is called mutable. For example, the processing of a probe tuple in HashJoin does not modify the list of build tuples for its key. Such operators whose states are immutable are called immutable-state operators. On the other hand, an input tuple to sort is added to the sorted list associated with its $\text{scope}$ (range of keys), thus it modifies the state. Such operators that have a mutable state are called mutable-state operators.

Notice that the execution of an operator can have more than one phase. For instance, a HashJoin operator has two phases, namely the build phase and the probe phase. The concept of mutability is with respect to a specific phase of the operator. In HashJoin, the states in the build phase are mutable, while the states in the probe phase are immutable. Reshape is applicable to a specific phase, and its state migration depends on the mutability of the phase. Table 1 shows a few examples of immutable-state and mutable-state operators.

### Table 1: Examples of physical operators based on state mutability. HB means hash-based and RB means range-based.

| Immutable-state operator | HashJoin (Probe phase), HB Set Difference (Probe phase), HB Set Intersection (Probe phase) |
|--------------------------|------------------------------------------------------------------------------------------------|
| Mutable-state operator   | HashJoin (Build phase), HB Group-by, RB Sort, HB Set Difference (Build phase), HB Set Intersection (Build phase), HB Set Union |

#### 5.2 Impact of mutability on state migration

Figure 10 shows how to handle state migration for operators when using the two load-transfer approaches discussed in Section 3.

**Figure 10: Operator state mutability and state migration.**

The state-migration process for immutable-state operators, as shown in branch (a) in Figure 10, involves replicating the skewed worker’s states at the helper, followed by a change in the partitioning logic. Thus, the tuples redirected from the skewed worker to the helper can use the state of their $\text{scope}$ at the latter. In contrast, the state-migration process is more challenging for mutable-state operators (branch (b)) because it is difficult to synchronize the state transfer and change of partitioning logic for a mutable state [38]. State-migration strategies that focus on such synchronization exist in the literature and will be briefly discussed in Section 5.3. As we show in Section 5.4, such a synchronization is not always possible.
Next, we discuss how to do state migration when using the two load-transfer approaches in mutable-state operators.

5.3 Mutable-state operators: split by keys

The SBK approach offloads the processing of certain keys in the skewed worker partition to the helper. Consider a group-by operator that receives tweets and aggregates the count of tweets per month. The skewed worker offloads the processing of a month (say, June) to the helper. There needs to be a synchronization between state transfer and change of partitioning logic so that the redirected June tuples arriving at the helper use the state formed from all June tuples received till then. In the case of group-by, this state is the count of all June tuples received by the operator. Existing work on state-migration strategies focuses on this synchronization. A simple way to do this synchronization is to pause the execution, migrate the state, and then resume the execution [9, 16, 48]. A drawback of this approach is that pausing multiple times for each iteration may be a significant overhead. Another strategy is to use markers [25]. The workers of the previous operator emit markers when they change the partitioning logic. When the markers from all the previous workers are received by the skewed and helper workers, the state can be safely migrated. Thus, skew handling in mutable-state operators using the "split by keys" approach can be safely done by using one of these state-migration strategies (branch (b1) in Figure 10).

5.4 Mutable-state operators: split by records

In this subsection, we use the SBR approach in mutable-state operators (branch (b2) in Figure 10). We show that the synchronization between state transfer and change of partitioning logic is not possible when using this approach and discuss its effects. Consider a sort operator with three workers, namely S1, S2, and S3, which receive range-partitioned inputs. The ranges assigned to the three workers are [0, 10], [11, 20], and [21, ∞]. As shown in Figure 11(a), S1 is skewed and S3 is its helper. The controller asks the previous operator to change its partitioning logic and send the tuples in [0, 10] to both S1 and S3 (Figures 11(b,e)). The synchronization of state migration and change of partitioning logic by the aforementioned state-migration strategies relies on an assumption that, at any given time, the partitioning logic sends tuples of a particular scope to a single worker only. When the tuples of [0, 10] are sent to both S1 and S3, this assumption is no longer valid. Worker S3 saves the tuples from the range [0, 10] in a separate sorted list (Figure 11(d)). Such a scenario where the val of a scope is split between workers is referred to as a scattered state.

This scattered state needs to be merged before outputting the results to the next operator. Now we explain a way to resolve the scattered state problem. When a worker of the previous operator finishes sending all its data, it notifies the sort workers by sending an END marker (Figure 11(d)). When S3 receives END markers from all the previous workers, it transfers its tuples in the range [0, 10] to the correct destination of those tuples, i.e., S1 (Figure 11(e,f)), thus merging the scattered states for the [0, 10] range.

We specify sufficient conditions for a mutable-state operator to be able to resolve the scattered state issue. The above approach of merging the scattered parts is suited for blocking operators such as group-by and sort, which produce output only after processing all the input data. Thus, the above approach can be used by mutable-state operators if they can 1) combine the scattered parts of the state to create the final state, and 2) block outputting the results till the scattered parts of the state have been combined.

6 RESHAPE IN BROADER SETTINGS

Our discussion about Reshape so far is based on several assumptions in Section 2 for simplification. Next we relax these assumptions.

6.1 High state-migration time

The state-migration time is assumed to be small till now. In this subsection, we study the case where this time could be significant.

Precondition for skew mitigation. In the discussion in Section 2, state migration is started immediately after skew detection. If the time to migrate state is more than the time left in the execution, the state migration is futile. Thus, the controller checks if the estimated state-migration time is less than the estimated time left in the execution and only then proceeds with state migration. The state-migration time can be estimated based on factors such as state-size and serialization cost [23, 59]. The time left in the execution can be estimated by monitoring the input data remaining to be processed and the processing speed [37] or by using the historical data [30].

Dynamic adaptation of τ. Suppose the adapted value of τ output by Algorithm 1 to be used in the next iteration is τn. The discussion in Section 4.3.2 assumes that the load transfer begins when the workload difference is around τn. This is possible only when the state-migration time is small. When the migration is significant, the load transfer will start when the workload difference becomes considerably greater than τn. In order to start the load transfer at τn (as assumed by Section 4.3.2), the skew has to be detected earlier. Thus, we adjust the skew detection threshold to τ′n, which is less than τn, such that the state migration starts when the workload difference is τ′n and ends when the workload difference is τn (Figure 12).

Formally, suppose t is the number of tuples processed by the operator per unit time, M is the estimated state-migration time, and ̂fS and ̂fH are the predicted workload percentages of S and H, respectively. The estimated difference in the tuples received by S and H during the state migration is (̂fS − ̂fH)∗t∗M. Therefore, given τn, the value of τ′n can be calculated as follows:

\[ τ′_n = τ_n - (\hat{f}_S - \hat{f}_H) \cdot t \cdot M. \]
6.2 Multiple helper workers

Till now we have assumed a single helper per skewer worker. Next, we extend Reshape to the case of multiple helpers.

Load reduction. The load reduction definition (Section 4.1) can be extended for $S$ and its helpers $h_1, \ldots, h_c$ as follows:

$$LR = \max_{w \in \{S, h_1, h_2, \ldots, h_c\}} (\sigma_w) - \max_{w \in \{S, h_1, h_2, \ldots, h_c\}} (\sigma'_w).$$

In the equation, $\sigma_w$ and $\sigma'_w$ are the sizes of the total input received by worker $w$ during the entire execution in the unmitigated case and mitigated case, respectively. Suppose $T$ is the total number of tuples received by the operator and $f_w$ is the actual workload percentage of a worker $w$. In the unmitigated case, $S$ receives the maximum total input among $S$ and its helpers, which is $f_S \cdot T$ tuples. In the ideal mitigation case, $S$ and its helpers have the same workload, which is the average of the workloads that they would have received in the unmitigated case. As discussed in Section 4.1, the ideal mitigation results in maximum load reduction denoted as:

$$LR_{\text{max}} = (f_S - \frac{\sum_{w \in \{S, h_1, h_2, \ldots, h_c\}} f_w}{n + 1}) \cdot T.$$

Choosing appropriate helpers. We examine the trade-off between the load reduction and the state-migration overhead to determine an appropriate set of helpers for $S$. Let $h_1, \ldots, h_c$ be $c$ helper candidates for $S$ in the increasing order of their workloads. From the definition above, increasing the number of helpers results in a higher $LR_{\text{max}}$, provided the average workload percentage reduces. However, increasing the number of helpers may result in higher state-migration time since more data needs to be transferred. Suppose $L$ is the number of future tuples to be processed by the operator at the time of skew detection. The estimated number of future tuples left to be processed by $S$ after state migration is $F = (L - M + t) \cdot f_S^e$. Increasing the number of helpers may increase the state-migration time ($M$) and thus decrease $F$, which means that there are fewer future tuples of $S$ to do load transfer. Thus, given a set of helpers, the highest possible load reduction after state migration is $\chi = \min(LR_{\text{max}}, F)$. As we add more helpers, $\chi$ initially increases and then starts decreasing. The set of helpers chosen right before $\chi$ starts decreasing are appropriate. Figure 13 illustrates an example. Let $W$ be the set of helper workers, which is initially empty. After adding $h_1$ to $W$, we have $LR_{\text{max}} < F$, thus $\chi = LR_{\text{max}}$. Then, we add $h_2$ to $W$, which decreases $F$, and $\chi = F$. Then, we add $h_3$ to $W$, which decreases $F$ further and causes $\chi$ to start decreasing. Hence, the final set of helpers for $S$ is $\{h_1, h_2\}$.

6.3 Unbounded data

The input has been assumed to be bounded till now. Next, we discuss a few considerations when the input is unbounded.

Load reduction and impact of $\tau$. In Section 4.1, the load reduction was calculated based on the total input received by the workers. For the unbounded case, the load reduction can be calculated based on the input received by the workers in a fixed period of time. The impact of $\tau$ on the load reduction holds for unbounded case too. A small value of $\tau$ results in high errors in workload estimation, which leads to a small load reduction. A large value of $\tau$ that takes too long to reach is not preferred in the unbounded case either. If a large $\tau$ delays mitigation, it can lead to back pressure, loss of throughput, and even crashing of data-processing pipelines. The latency of processing can increase, causing adverse effects on time-sensitive applications such as image classification in surveillance [32].

Merging scattered states. For bounded data, the scattered states in mutable-state operators were merged after the operator processed all the input. For unbounded data, the scattered states can be merged when the operator has to output results, e.g., when a watermark is received [12].

7 EXPERIMENTS

In this section, we present an experimental evaluation of Reshape using real and synthetic data sets on clusters.

7.1 Setting

Datasets and workflows. We used four datasets in the experiments. The first one included 180M tweets in the US between 2015 and 2021 collected from Twitter. The second dataset was generated using the DSB benchmark [22], which is an enhanced version of TPC-DS containing more skewed attributes, to produce record sets of different sizes ranging from 100GB to 200GB by varying the scaling factor. The third dataset was generated using the TPC-H benchmark [52] to produce record sets ranging from 50GB to 200GB. The fourth dataset was generated to simulate a changing key distribution during the execution. It included a synthetic table of 80M tuples and another table of 4, 200 tuples, and each table had two numerical attributes representing keys and values.

We constructed workflows of varying complexities as shown in Figure 14. Workflow $W_1$ analyzed tweets by joining them with a table of the top slang words from the location of the tweet. This workflow is used for social media analysis to find how often people use local slang in their tweets. The tweets were filtered on certain keywords to get tweets of a particular category. Workflow $W_2$ was constructed based on TPC-DS query 18, and it calculated the total count per item category for the web sales in the year 2001 by customers whose birth month $\geq 6$. Workflow $W_3$ read the Order's

[Figure 12: Adapt $\tau$ by considering the state-transfer time.]

[Figure 13: Choosing appropriate helpers.]

[Figure 14: Workflow $W_1$ analyzes tweets by joining them with a table of the top slang words from the location of the tweet. This workflow is used for social media analysis to find how often people use local slang in their tweets. The tweets were filtered on certain keywords to get tweets of a particular category. Workflow $W_2$ was constructed based on TPC-DS query 18, and it calculated the total count per item category for the web sales in the year 2001 by customers whose birth month $\geq 6$. Workflow $W_3$ reads the Order's]
table from the TPC-H dataset and filtered it on the orderstatus attribute before sorting the tuples on the totalprice attribute. Workflow W₄ joined the two synthetic tables on the key attribute. Figure 15 shows the distribution of the datasets of the datasets that may cause skew in the workflows. Figure 15a shows the frequency of tweets, used in W₁, based on the location attribute. Figure 15b shows the distribution of the Orders table on its totalprice attribute, used in W₃, for a 100GB TPC-H dataset. Figure 15c shows the distribution of the larger synthetic table in W₄ on the key attribute. Figures 15d-15f show the distribution of the three attributes of the sales table in W₂ used in the three join operations for a 1GB dataset.

Reshape implementation. We implemented Reshape⁴ on top of two open source engines, namely Amber [36] and Apache Flink (release 1.13). In Amber, we used its native API to implement the control messages used in Reshape. Unless otherwise stated, we set both r and η to 100. We used the mean model [51] to predict the workload of workers. In Flink, we used the busyTimeMsPerSecond metric of each task, which is the time ratio for a task to be busy, to determine the load on a task. We leveraged the mailbox of tasks (workers) to enable the control messages to change partitioning logic. The control messages are sent to the mailbox of a task, and these messages are processed with a higher priority than data messages in a different channel. Using these control messages, we implemented the two phases of the SBR load transfer approach on Flink as discussed in Section 3.1.

Baselines. For comparison purposes, we also implemented Flow-Join and Flux on Amber with a few adaptations. For Flow-Join, we used a fixed time duration at the start to find the overloaded keys. The workload on a worker was measured by its input queue size. For Flow-Join, after skew is detected, the tuples of the overloaded keys are shared with the helper worker in a round-robin manner.

For Flux, after skew is detected, the processing of an appropriate set of keys is transferred from the skewed worker to its helper. For both Reshape and the baselines, one helper worker was assigned per skewed worker, unless otherwise stated. Also, unless otherwise stated, Flux used a 2 second initial duration to detect overloaded keys. To be fair, when running Reshape, we also had an initial delay of 2 seconds to start gathering metrics and subsequent skew handling by Reshape.

All experiments were conducted on the Google Cloud Platform (GCP). The data was stored in an HDFS file system on a GCP dataproc storage cluster of 6 e2-highmem-4 machines, each with 4 vCPU's, 32 GB memory, and a 500GB HDD. The workflow execution was on a separate processing cluster of e2-highmem-4 machines with a 100GB HDD, running the Ubuntu 18.04.5 LTS operating system. In all the experiments, one machine was used to only run the controller. We only report the number of data-processing machines. The number of workers per operator was equal to the total number of cores in the data-processing machines and the workers were equally distributed among the machines.

7.2 Effect on results shown to the user
We evaluated the effect of skew and the different mitigation strategies on the results shown to the user. We ran the experiment on 48 cores (12 machines). California (location 6) produced the highest number of tweets (26M) in the tweet dataset. Arizona (location 4) and Illinois (location 17) produced 3.8M and 6.5M tweets, respectively. In the unmitigated case, the tuples of California (CA), Arizona (AZ), and Illinois (IL) were processed by workers 6, 4, and 17, respectively. We performed two sets of experiments, in which we mitigated the load on worker 6 processing CA tweets by using different helper workers. In the first set of experiments, we used worker 4 as the helper and monitored the ratio of CA to AZ tweets processed by the join operator. In the second set, we used worker 17 as the helper and monitored the ratio of CA to IL tweets processed by the join operator. The line charts in Figure 16 and 17 show the absolute difference of the observed ratio from the actual ratio as execution progressed. In the tweet dataset, the actual ratio of CA to AZ tweets was 6.85 and CA to IL tweets was 4.05.

No mitigation: When there was no mitigation, the CA, AZ, and IL tweets were processed at a similar rate as explained in Section 3.1. The observed ratio remained close to 1 till worker 4 was about to finish processing AZ tweets in Figure 16 and worker 17 was about to finish IL tweets in Figure 17. The observed ratio started to increase (absolute difference of observed ratio with actual ratio started to decrease) after that because worker 6 continued to process CA

⁴Reshape is available on Github (https://github.com/Reshape-skew-handle).
tweets. The actual ratio was observed near the end of execution (about 416 seconds) in the unmitigated case.

**Flux**: It used the SBK load-transfer approach. It had the limitation of not being able to split the processing of a single key over multiple workers. The skewed worker 6, apart from CA, was also processing the tweets from West Virginia. The processing of the tweets from West Virginia (about 600K) was moved to the helper worker by Flux. However, this did affect the observed ratio of tweets much.

**Flow-Join**: It used the SBR approach. The execution finished earlier because the approach mitigated the skew in worker 6. Flow-Join had two drawbacks. First, it did not perform mitigation iteratively. It changed its partitioning logic only once based on the heavy hitters detected initially. Second, it did not consider the loads on the helper and the skewed worker while deciding the portion of the skewed worker’s load to be transferred to the helper. It always transferred 50% of the load of the skewed worker to the helper. The observed ratio of tweets started increasing once skew mitigation started. It reached the actual ratio 198 seconds in Figure 16 and around 120 seconds in Figure 17. Due to the aforementioned drawbacks, the observed ratio of tweets continued to increase even after reaching the actual ratio because the skewed worker continued to transfer 50% of its load to the helper. The observed ratio continued to increase till it reached about 8.3 in Figure 16 (absolute difference = 1.5) and 6.2 in Figure 17 (absolute difference = 2.1). At this point, the execution was near its end and the ratio started to decrease to the actual final ratio.

**Reshape**: It used the SBR approach and could split the processing of the CA key with a helper worker. Reshape had the advantage of iteratively adapting its partitioning logic and considered the current loads on the helper and the skewed worker while deciding the portion of load to be transferred in the second phase (Section 3.2). Thus, Reshape kept the workload of the skewed worker and the helper at similar levels. In Figure 16 and 17, after the observed ratio reaches the actual ratio at about 120 seconds and 130 seconds, respectively, Reshape kept the observed ratio near the actual ratio.

### 7.3 Benefits of the first phase

We evaluated the benefits of the first phase in Reshape as discussed in Section 3.2. We followed a similar setting as in the experiment in Section 7.2 to monitor the ratio of processed tweets. There were two mitigation strategies used in this experiment. The first one was normal Reshape, with the two phases of load transfer. In the second strategy, we disabled the first phase in Reshape and just did load transfer using the second phase. The results are plotted in Figure 18 and 19.

### 7.4 Effect of heavy-hitter keys

California (location 6) produced the highest number of tweets (26M) and was a heavy-hitter key in the tweet dataset. We present the results for the mitigation of the skewed worker that processed the California key.

**Load balancing ratio**: The load balancing ratio at a moment during the execution is calculated by obtaining the total counts of tuples allotted to the skewed worker and its helper till that moment, and dividing the smaller value by the larger value. We periodically recorded multiple load balancing ratios during an execution and calculated their average to get the average load balancing ratio for an execution. A higher ratio is better because it represents a more balanced workload between the skewed worker and its helper.

The average load balancing ratio for the skewed worker that processed the California key and its helpers is plotted in Figure 20. A higher ratio is better because it represents a more balanced workload between the skewed worker and its helper worker. We ran
the experiments on three settings by varying the number of cores up to 56 (on 14 machines), which was the total number of distinct locations.

![Figure 20: Evaluating different methods of handling heavy-hitter keys in W1 using tweets. The three Flow-Join bars correspond to the initial delay of 2, 4, and 8 seconds.](image)

Flux: It had the limitation of not being able to split the processing of a single key over multiple workers. Thus, the skewed worker processed the entire California input. The skewed worker was also processing another key with only a few hundred thousand tuples, which was moved to the helper when skew was detected. Flux had a low average load balancing ratio of about 0.06.

Flow-Join: Its main drawback was the inability to do mitigation iteratively. It changed its partitioning logic once based on the heavy-hitters detected initially. The longer it spent to detect heavy-hitters with a higher confidence, the less was the amount of future tuples left to be mitigated for finite datasets. We varied the initial duration used by Flow-Join to detect heavy-hitters from 2 seconds to 8 seconds. When the initial time spent was 2 seconds, the average load balancing ratio was about 0.85 and the final counts of tuples processed by the skewed and helper workers were approximately 14M and 12M, respectively. On the other hand, when the duration was 8 seconds, the ratio was about 0.6 and the final counts were approximately 17M and 9M, respectively. Flow-Join was able to reduce the execution time of W1 on 48 cores from 416 seconds to 302 seconds, when the initial detection duration was 2 seconds.

Reshape: It split the processing of the California key with a helper worker. Reshape had the advantage of iteratively changing its partitioning logic according to input distribution using fast control messages. Thus, the skewed and helper workers ended up processing almost similar amounts of data and the average load balancing ratio was about 0.92. The execution time was reduced by 27%. In particular, Reshape was able to reduce the execution time from 416 seconds to 302 seconds, by mitigating the skew in W1 running on 48 cores.

### 7.5 Effect of latency of control messages

To evaluate the effect of the latency of control messages on skew handling by Reshape, we purposely added a delay between the time a worker receives a control message and the time it processes the message. Figure 21 shows the result of varying the simulated delay from 0 second (i.e., the message is processed immediately) to 15 seconds on the mitigation of W1 on 48 cores. The figure shows the average load balancing ratio for the two pairs of skewed and helper workers processing the locations of California (location 6) and Texas (location 48), which had the highest counts of tweets.

![Figure 21: Effect of control message delay (W1 on tweets).](image)

**Impact on responsiveness of mitigation:** As the control message delivery became slower, the delay between the controller sending a message and the resulting change in partitioning logic increased. Consider the example where the controller detected a workload difference of 350 between the skewed worker and the helper worker and sent a message to start the first phase. In the case of no delay in control message delivery, the helper worker reached a similar workload as the skewed worker within 10 seconds. In case of a delayed delivery, the workload difference continued to increase and got larger than 350 before the first phase was started. For example, when there was a 5-second delay, the workload difference was at 300 after 10 seconds of sending the message.

**Impact on load balancing.** The latency in control messages affected the load sharing between skewed and helper workers. In the case of no delay, the two workers had almost similar loads and the average load balancing ratio was about 0.94 as shown in Figures 21a and 21b. As the delay increased, the framework was slow to react to the skew between workers, which resulted in imbalanced load-sharing. In the case of a 15-second delay, the average load balancing ratio reduced to about 0.45. Thus, low-latency control messages facilitated load balancing between a skewed worker and its helper.

### 7.6 Benefit of dynamically adjusting \( \tau \)

We evaluated the effect of the dynamic adjustment of \( \tau \) on skew mitigation in W1 by Reshape on 48 cores. We chose different values of \( \tau \) ranging from 10 to 2,000, and did experiments for two settings. In the first setting, \( \tau \) was fixed for the entire execution. In the second setting, \( \tau \) was dynamically adjusted during the execution. The mean model estimated the workload of a worker as its expected number of tuples in the next 2,000 tuples and the preferred range of standard error (Section 4.3.2) was set to 98 to 110 tuples. We allowed up to three adjustments during an execution. Whenever \( \tau \) had to be increased, it was increased by a fixed value of 50. We calculated the average load balancing ratios for the workers processing the California and Texas keys and divided them by the total number of mitigation iterations during the execution. This resulted in the metric of average load balancing per iteration, shown in Figure 22. A higher value of this metric is better because it represents a more balanced workload of skewed and helper workers in fewer iterations.

Let us first consider the cases where \( \tau \) was dynamically adjusted to an increased value. Setting \( \tau \) to a small value of 10 resulted in a large number of iterations, i.e., 41, in the fixed \( \tau \) setting. In the dynamic \( \tau \) setting, the controller observed that the standard error at the beginning of the second phase was greater than 110 and increased \( \tau \). Consequently, the number of iterations decreased to 14, which resulted in a substantial increase in the metric of average load balancing per iteration. For the cases of \( \tau = 50 \) and 100 in the
fixed setting, the average load balancing per iteration increased with $\tau$ because the number of iterations decreased. The dynamic setting slightly decreased the iteration count in these cases.

Now let us consider the case where $\tau$ remained unchanged or decreased as a result of dynamic adjustment. When $\tau = 500$, the standard error was in the range [98, 110]. Thus, the dynamic adjustment did not change $\tau$. When $\tau = 1000$ in the fixed setting, the mitigation started late and the workload of skewed and helper workers were not balanced. The mitigation was delayed even more for $\tau = 1500$ and 1800 in the fixed setting and the mitigation did not happen for $\tau = 2000$. In the dynamic setting for the cases of $\tau = 1000$, 1500, 1800, and 2000, the controller observed that the standard error went below 98 when the workload difference was about 700. Thus, the controller reduced $\tau$ to 700. The advantage of dynamically reducing $\tau$ was that it automatically started mitigation at an appropriate $\tau$, even if the initial $\tau$ was very high.

### 7.7 Effect of different levels of skew

We evaluated the load balancing achieved by Reshape for different levels of skew. We used $W_2$ for this purpose. The data distributions in Figures 15d-15e show that the join on item_id was highly skewed and the join on date_id was moderately skewed. We evaluated the load balancing achieved for these two join operators. We scaled the data size from 100GB to 200GB. Meanwhile, we scaled the number of cores from 40 to 80 and did the experiments in each configuration.

![Figure 23: Effect of different levels of skew ($W_2$ on DSB data). Each candlestick body represents the 25th to 75th percentile.](image)

Figure 23 shows the candlestick charts of the average load balancing ratios for the top five skewed workers from each of the two joins. For the highly skewed join on item_id, the skew was detected early, and there was enough time to transfer the load of the skewed workers to the helper workers. The 25th and 75th percentiles of the average load balancing ratios remained above 0.6 for all the configurations. The median of the ratios was more than 0.77. This result shows that Reshape was able to mitigate the skew and maintain comparable workloads on the skewed and helper workers when both the input and processing power were scaled up. The join on date_id had only a moderate skew, which resulted in a delayed detection of a few of its skewed workers. Due to the delayed detection, there were fewer future tuples of skewed workers to be transferred to the helpers. Thus the ratios for the join on date_id were lower than that for the join on item_id. The performance of Reshape was also shown by the reduction in the execution time. Specifically, in the case of 40 cores, the mitigation reduced the execution time of $W_2$ from 267 seconds to 243 seconds. In the case of 80 cores, the mitigation reduced the time from 335 seconds to 269 seconds.

### 7.8 Effect of changes in input distribution

We evaluated how load sharing was affected when the input distribution changed during the execution. We used the synthetic dataset and workflow $W_4$ running on 40 cores. Both tables in the dataset had 42 keys. The first table contained 4, 200 tuples uniformly distributed across the keys. The second table contained 80M tuples and was produced by the source operator at runtime. We fixed worker 0 and worker 10 as the skewed and helper worker, respectively. We altered the load on key 0 and 10, which were processed by worker 0 and 10 respectively. Specifically, for the first 20M tuples, 80% was allotted to the key 0 and the rest 20% was uniformly distributed among the remaining keys. For the next 60M tuples, 60% was allotted to the key 0, 20% to key 10, and the rest was uniformly distributed. Figure 24 shows the ratio of the workloads of the helper worker 10 to the skewed worker 0 as time progressed. We used $\tau = 2,000$ to clearly show the effects of changing distributions.

![Figure 24: Effect of changes in input data distribution on load sharing ($W_4$ on the synthetic dataset).](image)
started another iteration of mitigation and adjusted the partitioning logic according to the new input distribution. As a result, the ratio of the workloads of the workers remained close to 1.

### 7.9 Metric-collection overhead

We evaluated the metric-collection overhead of Reshape on the workflow $W_2$. We scaled the data size from 100GB to 200GB. Meanwhile, we scaled the number of cores from 40 (on 10 machines) to 80 (on 20 machines) and did the experiments in each configuration. We disabled skew mitigation and executed $W_2$ with and without metric collection to record the metric-collection overhead. As shown in Figure 25, the overhead was around 1-2% for all the configurations.

![Figure 25: Metric-collection overhead ($W_2$ on DSB data).](image)

### 7.10 Performance of Reshape on sort

To evaluate its generality to other operators, we implemented Reshape for the sort operator. We used the workflow $W_3$ for this experiment. The Orders table was range-partitioned on its totalPrice attribute. Table 2 lists the various percentile values of the average load balancing ratio for the skewed workers that received more than 3.5M tuples in the unmitigated case (Figure 15b). We scaled the data size and number of cores simultaneously from 50GB on 20 cores to 200GB on 80 cores, and did the experiment in each configuration.

| # workers | $P_0.1$ | $P_{25}$ | $P_{50}$ | $P_{75}$ | $P_{90}$ |
|-----------|---------|---------|---------|---------|---------|
| 20        | 0.90    | 0.92    | 0.93    | 0.935   | 0.95    |
| 40        | 0.84    | 0.87    | 0.89    | 0.90    | 0.91    |
| 60        | 0.83    | 0.85    | 0.90    | 0.91    | 0.92    |
| 80        | 0.83    | 0.84    | 0.86    | 0.87    | 0.90    |

Table 2: Average load balancing ratios when Reshape is applied on sort ($W_3$ using the TPC-H data).

As the number of cores increased, the $25^{th}$ and $75^{th}$ percentiles of the average load balancing ratios remained close to 0.9. This result shows that the skewed and helper workers had balanced workloads when both the input and processing power were scaled up. The consistent performance of Reshape was also shown by about 20% reduction in the execution time. Specifically, in the case of 20 cores, the time reduced from 789 seconds to 643 seconds. In the case of 80 cores, the time reduced from 809 seconds to 667 seconds.

### 7.11 Effect of multiple helper workers

We evaluated the load reduction achieved when multiple helper workers are assigned to a skewed worker. The experiment was done on $W_1$ running on 48 cores. The most skewed worker among the 48 workers received about 27M tuples in the unmitigated case. We allotted different numbers of helpers to the skewed worker and calculated the load reduction. We set the build hash-table in each worker to have 10,000 keys, so that the state size became significant and the state-migration time was noticeable.

![Figure 26: Effect of multiple helper workers ($W_1$ on tweets).](image)

The results are plotted in Figure 26. When a single helper was used, the state migration happened in 17 seconds. The skewed worker transferred about half of its total workload to the helper, resulting in a load reduction of 13M tuples. When 2 helpers were used, the skewed worker transferred about two thirds of its tuples to the two helpers (about 9M each). With more helpers, the state-migration time also increased. For 8 helpers, the state-migration time was about 26 seconds. Thus, there were fewer future tuples left, which resulted in a small increase in the load reduction. For 16 helpers, the state-migration time became 32 seconds and the load reduction decreased to 19.7M. For 24 helpers, the state-migration time was 39 seconds and the load reduction decreased to 19M.

### 7.12 Performance of Reshape on Flink

We implemented Reshape on Apache Flink and executed $W_1$ on 40, 48, and 56 cores. A worker was classified as skewed if its busyTimeMsPerSecond metric was greater than 80%. Figure 27 shows the average load balancing ratio for the workers processing the California and Texas tweets. The ratio was about 0.9, which means that the skewed and helper workers had similar workloads throughout the execution. For the 48-core case, the final counts of tuples processed by the skewed and helper workers for California were 13M and 14M, respectively. The final counts of tuples processed by the two workers for Texas on 48 cores were 10M each. The execution time decreased as a result of the mitigation. For example, for the 48-core case, the execution time decreased from 407 seconds to 320 seconds.

![Figure 27: Mitigation by Reshape on Flink ($W_1$ on tweets).](image)

(a) California data. (b) Texas data.

### 8 CONCLUSIONS

In this paper we presented a framework called Reshape that adaptively handles partitioning skew in the exploratory data analysis setting. We presented different approaches for load transfer and analyzed their impact on the results shown to the user. We presented an analysis about the effect of the skew-detection threshold on mitigation and used it to adaptively adjust the threshold. We generalized Reshape to multiple operators and broader execution settings. We implemented Reshape on top of two big data engines and presented the results of an experimental evaluation.
