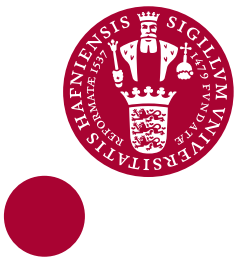


Towards Low-Latency Batched Stream Processing by Pre-Scheduling

Jin, Hai; Chen, Fei; Wu, Song; Yao, Yin; Liu, Zhiyi; Gu, Lin; Zhou, Yongluan

Published in:
IEEE Transactions on Parallel and Distributed Systems

DOI:
10.1109/TPDS.2018.2866581

Publication date:
2019

Document version
Peer reviewed version

Towards Low-Latency Batched Stream Processing by Pre-Scheduling

Hai Jin, *Senior Member, IEEE,* Fei Chen, Song Wu, *Member, IEEE,*
Yin Yao, Zhiyi Liu, Lin Gu, and Yongluan Zhou

Abstract—Many stream processing frameworks have been developed to meet the requirements of real-time processing. Among them, batched stream processing frameworks are widely advocated with the consideration of their fault-tolerance, high throughput and unified runtime with batch processing. In batched stream processing frameworks, straggler, happened due to the uneven task execution time, has been regarded as a major hurdle of latency-sensitive applications. Existing straggler mitigation techniques, operating in either reactive or proactive manner, are all post-scheduling methods, and therefore inevitably result in high resource overhead or long job completion time. We notice that batched stream processing jobs are usually recurring with predictable characteristics. By exploring such a heuristic, we present a pre-scheduling straggler mitigation framework called Lever. Lever first identifies potential stragglers and evaluates nodes' capacity by analyzing execution information of historical jobs. Then, Lever carefully pre-schedules job input data to each node before task scheduling so as to mitigate potential stragglers. We implement Lever and contribute it as an extension of Apache Spark Streaming. Our experimental results show that Lever can reduce job completion time by 30.72% to 42.19% over Spark Streaming, a widely adopted batched stream processing system and outperforms traditional techniques significantly.

Index Terms—stream processing; recurring jobs; straggler; scheduling; data assignment

1 Introduction

With the vast involvement of streaming big data in many applications (e.g., stock market data, sensor data, social network data, etc.) [1] [2], quickly mining and analyzing such data is becoming more and more important. There is a recent trend in adapting batch processing systems, such as MapReduce [3] and Spark [4], to handle streaming data by putting the streams into micro-batches and treating the workloads as a continuous series of small jobs. Examples of such systems include HOP [5], Comet [6], HStreaming [7] and Spark Streaming [8]. Figure 1 illustrates a simplified model of the data handling pipeline in such systems. In this model, the *batching module* receives and divides data streams into batches, which are then put into the *batch queue*. The *processing module* schedules tasks for processing according to the bulk synchronous parallel (BSP) model.

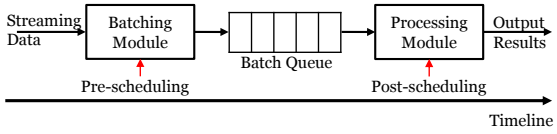


Fig. 1: A simplified data processing pipeline in a batched stream processing system

Batched stream processing systems become popular because not only they provide a unified programming model for processing batch data and streaming data, but also they can leverage the fault tolerance and high throughput properties of the batch processing frameworks [9]. However, in comparing to record-at-a-time stream processing systems, such as Storm [10], and Naiad [11], batched steam processing systems often suffer from high processing latency. Therefore, the fundamental challenge of building a batched stream processing system is to minimize the processing latency of each micro-batch. This problem has recently attracted the community's interest, and some efforts, such as Drizzle [12], have been focused on minimizing the coordination overhead of processing the micro-batches, including the cost incurred by task scheduling and barriers between different processing stages.

In this paper, we focus on an critical issue, namely the straggler problem, where a subset of tasks straggling behind and significantly affecting the job completion time. The straggler problem is a well-known essential problem in parallel processing systems. Stragglers can occur for many reasons, including hardware heterogeneity [13] [14], data skew [15] [16], hardware failures [17], energy efficiency [18] [19], resource contention and various OS effects. As reported in [20] and [21], in the production clusters at Facebook and Microsoft Bing, straggler tasks are on average 8 times slower than the median task in the same job.

Although stragglers in batched stream processing systems have the same definition with batch processing systems, they have different characteristics such as latency sensitive and straggler locality. In comparing to large batch processing, the straggler problems in micro-batch processing are more severe and harder to tackle. First of all, unlike processing large batches, where there are multiple waves of tasks to be scheduled, the number of tasks of each micro-batch processing is limited. Therefore, once a straggler occurs, scheduler has little opportunity to amortize the influence of straggler. Second, batch processing systems usually adopt the well-known BSP model, where a barrier is placed at the end of each processing stage and all the parallel tasks within the same stage have to synchronize at the barrier. As a batched stream processing system should handle many continuously arriving micro-

- *H. Jin, F. Chen, S. Wu, Y. YAO, Z. Liu and L. Gu are with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: wusong@hust.edu.cn*
- *Y. Zhou is with the Department of Computer Science, University of Copenhagen.*

batches, if the execution time of the straggling tasks exceed the batch interval, it would affect not only the latency of the current micro-batch, but also the queueing latency of the subsequent ones, which can become unacceptably high. Furthermore, the actions of handling stragglers have to be carried out very quickly so that the total task (re-)scheduling and processing time should not exceed the batch interval. This is especially challenging if we use small, say sub-second, batch intervals. For example, the wait-speculate-re-execute paradigm adopted in many existing solutions [3] [22] [15] would be undesirable, because the processing time of the micro-batches can be too short to afford the waiting time to detect the stragglers, not to mention the additional latency incurred by relocating the data and re-executing the straggling tasks.

We argue that the fundamental problem of using the existing straggler mitigation solutions for micro-batch processing is that they detect (or predict) stragglers and re-schedule stragglers too late in the data handling pipeline, which is after the batching module has already dispatched the data into the batch queue and the processing module has started processing the data (recall Figure 1). The re-scheduling actions are carried out during the task execution period, hence it would inevitably increase the processing time of the micro-batches. Furthermore, as the data has already been dispatched, re-scheduling would inherently incur expensive data relocation. Such overhead would become significant in micro-batch processing due to the short processing time of each micro-batch. We refer to this type of methods as *post-scheduling* techniques.

To address the problem, we propose a new *pre-scheduling* framework, called *Lever*, which predicts stragglers and makes timely scheduling decisions to minimize processing latency. Lever utilizes the historical processing statistics to predict the potential stragglers in the current batch and makes proactive pre-scheduling decisions to prevent stragglers. More importantly, Lever makes the pre-scheduling decisions before the batching module dispatches the data. Therefore, the pre-scheduling actions would not incur any data relocation. As the scheduling is done while the data are being batched, it would not increase the processing time of the micro-batch.

We implemented Lever in Spark Streaming, which is contributed to the open source community as an extension of Apache Spark Streaming. In summary, this paper makes the following technical contributions:

- We perform a thorough analysis of the behaviors of existing straggler mitigation methods and identify the problems of applying them in batched stream processing systems. We find that these methods are all post-scheduling methods and fall short when dealing with stragglers in micro-batch jobs.
- To better mitigate stragglers when processing micro-batches, we present Lever, a pre-scheduling framework for handling straggler problems in batched stream processing systems. Lever predicts and re-schedules straggling tasks before the data dispatching and task processing to avoid the latency incurred by the post-scheduling methods. Furthermore, by mitigating the stragglers, Lever can significantly reduce the processing latency of micro-batches.
- We propose various techniques to realize the new pre-scheduling framework. We design a method to predict the potential stragglers before the execution of the tasks by using the historical statistics, which takes into account the variations of data streaming rates. In addition, we adopt the Iterative Learning Control model to estimate the capacity of the computing nodes. Finally, we propose a capacity-aware data assignment strategy to pre-schedule input data.
- We have implemented Lever on Spark Streaming, and contributed it as an extension of Spark Streaming to the open source community. Extensive experiments using both real and synthetic data show that Lever can mitigate stragglers efficiently and improve the performance of stream applications by 30.72% to 42.19% over Spark Streaming.

The rest of this paper is organized as follows. In Section II, we introduce the background and analyze the straggler problems in batched stream processing system. Section III describes the pre-scheduling strategy and the design of Lever. Section IV presents the implementation of Lever. We evaluate the performance of our system in Section V. Section VI briefly surveys the related works. Finally, Section VII concludes this paper.

2 Background and Motivation

In this section, we first introduce the background of batched stream processing and stragglers. Then we analyze in detail the problems of using traditional straggler mitigation strategies in batched stream processing. We also briefly analyze the characteristics of the recurring batched stream jobs. Finally, we outline the challenges in realizing the proposed pre-scheduling framework.

2.1 Background

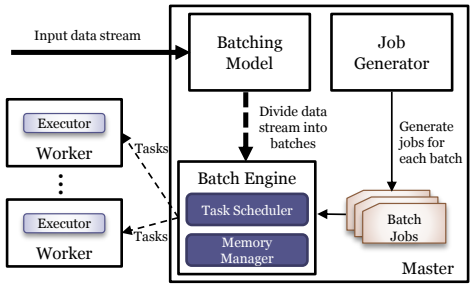


Fig. 2: Principles of batched stream processing system

Batched stream processing system treats a streaming computation as a series of deterministic batch computations on small time intervals [8]. As shown in Figure 2, a batched stream processing system receives input data and divides the continuous data streams into a series of so-called micro-batches according to the specified batch interval (a time interval set according to the application's latency requirements). A number of jobs are generated for each batch, which would be periodically submitted to the batch processing engine, such as MapReduce or Spark. Such a batched stream processing system automatically inherits the advanced features of the underlying batch processing systems, such as fault tolerance, data-locality aware scheduling, load balancing, etc.

2.2 Problem Analysis

Existing methods for straggler mitigation are post-scheduling. In the following, we analyze four representative approaches. They are Speculative execution [3], SkewTune [15], Dolly [20] and Wrangler [21], on behalf of reactive approach with replication, reactive

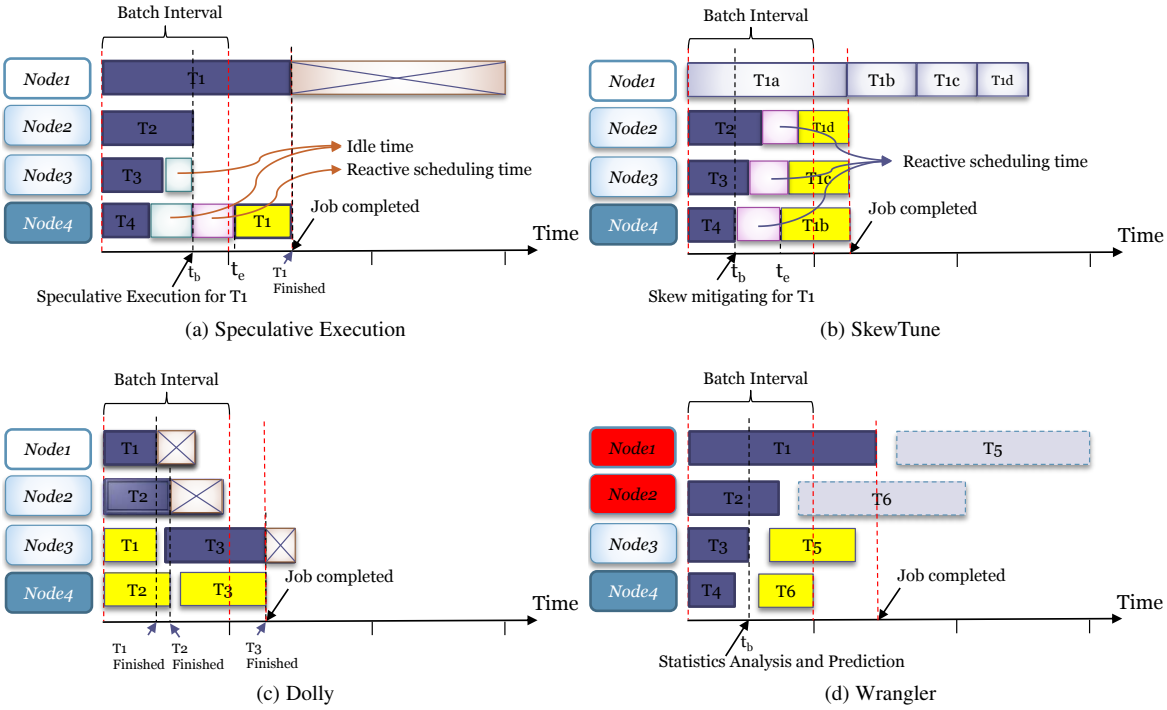


Fig. 3: Straggler mitigation under different strategies

approach without replication, proactive approach with replication and proactive approach without replication respectively.

Speculative execution [3] is a widely used reactive approach with replications. It employs a wait-speculate-re-execute mechanism. Speculative execution marks slow running tasks as stragglers and reacts by re-launching multiple copies of them. As soon as one of these copies finishes execution, the rest are terminated. Taking Figure 3(a) as an example, at t_b, T1 is marked as stragglers. Then speculative execution launches a copy of T1 on $Node_4$. It migrates the data block of Task T1 to $Node_4$ and starts executing it at t_e. As we can see, this approach incurs long idle time and reactive scheduling time. It introduces inefficiency for micro-batch jobs in two aspects: (1) a task must run for a significant amount of time before it is identified as a straggler, leading to delayed straggler detection; (2) the reactive procedure is time-consuming in the context of small batch intervals.

SkewTune [15] is a reactive approach without replications. Once one node has an available slot, SkewTune begins to detect data skew and identify stragglers according to each task's remaining time. As shown in Figure 3(b), when task T_4 is completed at t_b, SkewTune's detection is triggered and T_1 is identified as straggler. Then, T_1 is selected for skew mitigation due to its long remaining progress. After that, SkewTune scans T_1's remaining input data and repartitions its remaining workload into T_{1b}, T_{1c} and T_{1d}, which are migrated to nodes $Node_4$, $Node_3$ and $Node_2$, respectively. Before SkewTune begins its actions, task T_1 has been already slowed down. Therefore, SkewTune belongs also the wait-and-speculate paradigm. Furthermore, the cost of reactive re-scheduling would be too high for the short-running micro-batch jobs.

Dolly [20] is a proactive strategy, which launches multiple clones of each task and only uses the result of the clone that finishes first. As shown in Figure 3(c), Dolly launches two clones of Task T_1 and T_2 in $Node_1$, $Node_3$ and $Node_2$, $Node_4$ respectively. After T_1 and T_2 complete, Dolly launches two clones of T_3 in $Node_3$ and $Node_4$ respectively. Although multiple clones for each task are spawned, only one of them is effective, incurring significant waste of resources.

Wrangler [21] is a proactive strategy without replication. It predicts stragglers using machine learning models and makes scheduling decisions based on prediction. For example, in Figure 3(d), Wrangler analyzes the statistics and makes a prediction about stragglers at t_b. $Node_1$ and $Node_2$ are identified as stragglers. Then Wrangler re-schedules and migrates the unprocessed tasks, i.e. T5 and T6, from $Node_1$ and $Node_2$ to $Node_3$ and $Node_4$ respectively. Although this approach can predict possible stragglers earlier than the previous reactive approaches, it still needs to wait until some nodes show some indications of straggling (e.g. high cpu utilization, heavy memory access, network congestion). Note that a micro-batch job often only consists of a limited number of tasks, which leaves little room for Wrangler to do the re-scheduling.

In summary, existing post-scheduling approaches for batch processing systems fall short when processing short-running micro-batch jobs. Reactive approaches act after stragglers have occurred, i.e. when tasks have already been slowed down. Replication-based proactive approaches incur extra resources and increase the system load. Proactive approaches without replication also need to migrate data block at runtime. Consequently, it is necessary to design a straggler mitigation strategy for batched stream processing systems that can act as early as possible, without incurring much extra resource overhead.

2.3 Recurring Batched Stream Jobs

Batched stream jobs are periodic in nature, hence are typically recurring with stable code and similar data features. Many char-

acteristics, such as application logic, stragglers, and resource utilization of the processing of a batch are statistically similar to the execution of the previous batches of the same stream [23] [24] [25]. In practice, the history of the prior runs of such recurring jobs can be used to estimate the task durations and predict stragglers.

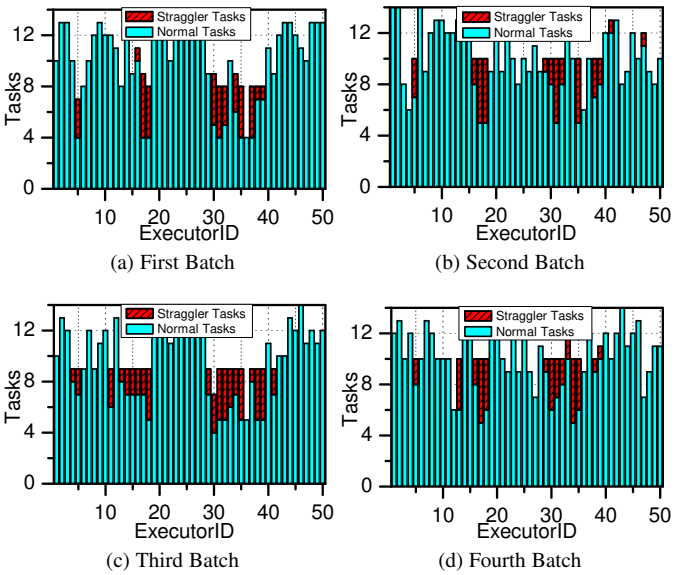


Fig. 4: Task statistics in four continuous batches. If one executor has straggler tasks in previous batch, there is a great probability that this executor will produce straggler tasks again in the next batch.

Figure 4 is a straggler statistical analysis which counts the number of straggler tasks and normal tasks in each executor from a real application in Tencent production clusters. This application is a typical real-time stream application monitoring advertisement click traffic based on Spark Streaming. It runs in a multi-tenant environment with YARN(Hadoop-2.2.0) as the resource management platform. The whole trace of this application can be accessed at https://github.com/u2009cf/Tencent-Cluster-Trace. Figure 4(a), (b), (c) and (d) are four continuous batches. From Figure 4, we can see that if one executor has straggler tasks in previous batch, there is a great probability that this executor will produce straggler tasks again in the next batch. we draw a conclusion that stragglers always occur in continuous batches in the same executor. We call this phenomenon as *straggler locality*. By statistically analyzing one day's logs of the Tencent application, we can get the result that the probability of which an executor which has straggler tasks in this batch will produce straggler tasks in next batch is 83.21%.

Considering that stragglers can often be accurately predicted in recurring jobs and job input data can be assigned in designated locations in advance, we can try to mitigate stragglers by performing pre-scheduling, where data dispatching is carried out before task execution by exploring the statistical similarity in short batched stream jobs.

2.4 Challenges

To realize a pre-scheduling framework, we need to address the following three challenges:

Challenge 1: How to identify potential stragglers?

Pre-scheduling needs to know which nodes will be the stragglers in the next batch. Unlike traditional post-scheduling approaches, which identify stragglers of a job based on the running time of the tasks of this job, pre-scheduling takes actions in the batching module without task execution information. Pre-scheduling can only use the historical execution information of recurring jobs. But only analyzing historical information is not enough due to the runtime changes of streaming data. Increasing the accuracy of identifying potential stragglers is a prerequisite step.

Challenge 2: How to determine the node capacity?

Traditional post-scheduling methods monitor the resource utilization at runtime and can re-schedule straggler tasks to those nodes which have free slots. However, in pre-scheduling solutions, this is not the case because we make pre-scheduling decisions before running the tasks. We need to define a metric to estimate the potential free slots of a node in the future and to determine how much data we should pre-schedule to the targeted nodes. How to determine a node's capacity has a great influence on the performance of pre-scheduling.

Challenges 3: How to conduct data reassignment efficiently?

To pre-schedule input load as evenly as possible to eliminate stragglers, we need to conduct data reassignment to achieve load balancing. However, batched stream processing system aggregates stream data into a block for a fixed time interval called the *blockInterval*. As the input rate varies over time, blocks may be unequal in size, leading to imbalanced load. Furthermore, current batched stream system assumes that the workload remains constant over time and the executors all have equal processing capacity. Data reassignment in pre-scheduling framework should adjusts the block sizes and quantities to match each node's capacity.

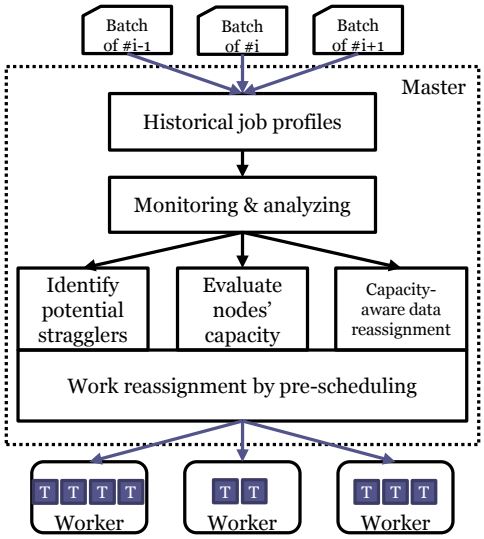


Fig. 5: Architecture of Lever

3 System Design

In this section, we present the design of Lever. Lever is API-compatible with Spark Streaming, providing the migration transparency and developer transparency. Considering the main objective is to minimize latency, Lever is designed to be lightweight and to be able to make agile scheduling decisions. We begin with an

overview of the system architecture, followed by further system details.

3.1 System Overview

Figure 5 overviews the architecture of Lever. Lever periodically collects and analyzes the historical job profiles of the recurring micro-batch jobs. Based on such information, Lever pre-schedules the data through three main steps, i.e. identify potential stragglers, evaluate node capacity and reassign data being aware of capacity. Firstly, by comparing each node's task finish time in the previous batch, Lever can determine the initial state of each node. Lever also monitors the changes of the input rate. With these two pieces of information, Lever conducts the state transition to predict which nodes will behave as stragglers in next batch (for Challenge 1). Secondly, based on the fact that micro-batch processing jobs are repetitive and periodic, Lever adopts the Iterative Learning Control (ILC [26]) model, which is designed for tracking control of the systems working in a repetitive mode, to estimate node capacities (for Challenge 2). Finally, Lever partitions the large tasks in straggler nodes and adapts the block sizes and quantities according to each node's capacity (for Challenge 3).

We show the timing of Lever's actions in Figure 6, where we differentiate pre-scheduling and post-scheduling techniques according to their scheduling timing. It can be seen that the post-scheduling techniques, such as Dolly, Wrangler, Speculation and SkewTune, take actions during task processing, which would inevitably increase the task processing latency. On the contrary, Lever takes its action before task execution. As shown in Figure 6, Lever collects the needed information of the previous micro-batch's execution during batch interval between t_1 and t_2. Then, during t_2 and t_3, Lever dispatches the input data based on the pre-scheduling plan, and tasks would be scheduled according to data locality. In this way, Lever would not incur too much data movement when processing tasks. We detail each step of Lever in the following.

3.2 Potential Stragglers Identification

Lever predicts stragglers in the next batch according to the historical information of the recurring jobs as well as the load fluctuation of each node. The first step is to determine the initial stragglers. When the tasks of the last batch are completed, Lever collects and analyzes the statistics of the task execution in each node. The node i's finish time (NFT_i) is defined as the time from job submission to when the last task is completed in this node. Then, Lever sorts node list according to NFT_i in the descending order. By following the experiences in previous work [3], we classify nodes into three categories according to the locations of the node list. Nodes before the first quartile are put into the *straggler group*, while those after the third quartile are in the *faster group*. The remaining nodes are categorized as the *median group*. Nodes in faster group are those nodes which are eligible to provide assistance. Nodes in straggler group are those nodes which need to be helped. And nodes in median group will do nothing.

The classification identifies the stragglers in the last batch. However, as data streaming rates vary over time and load can change across different batches, the state of each node may change between two consecutive batches. Therefore, we should carefully derive the possible state transitions to accurately identify the stragglers as follows. First, Lever calculates the median finish time of the *straggler group*, *median group* and *faster group* respectively. We denoted these values by t_{ftos}, t_{ftom}, t_{ftof} respectively. Second, Lever uses two degradation ratios *FTM* and *MTS*. *FTM* is defined as $\frac{t_{ftom}}{t_{ftof}}$, which means that for node i in the *faster group*, if NFT_i has increased by FTM, it will be moved from the *faster group* to the *median group* and vice versa. Similarly, *MTS* is defined as $\frac{t_{ftos}}{t_{ftom}}$, which means that for node i in the *median group*, if NFT_i has increased by MTS, it will be moved from the *median group* to the *straggler group* and vice versa. We have shown straggler's state transition in Figure 7.

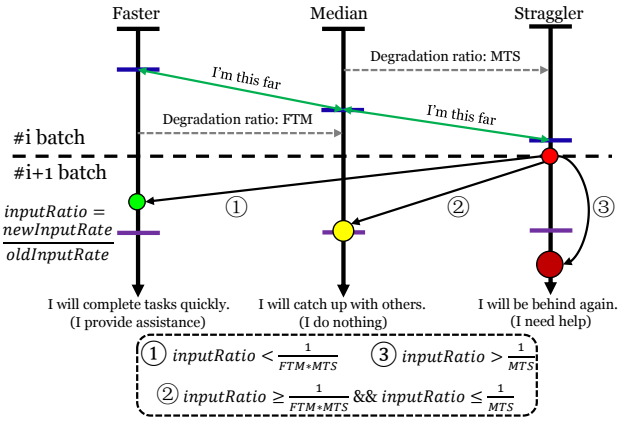


Fig. 7: An example for straggler's state transition

Based on the transition graph in Figure 7, Lever takes the load fluctuation observed on each node to predict state transition and identify stragglers. Due to the load fluctuation, it is possible that some stragglers in the last batch may receive less data and hence become faster in the current batch. Similarly, the faster nodes may possibly become stragglers in current batch. We use *inputRatio*, defined as the ratio between the new input data rate of current batch and the old one of the last batch, to evaluate the changes of stream load. According to the transition graph, we define the transition rules listed in Table 1. By applying these rules, Lever finally identifies the state of each node for the current batch.

TABLE 1: Transition rules for identifying stragglers

Initial State	Transition Conditions(inputRatio)	Final State
Straggler	(1/MTS,+∞)	Straggler
	[1/(FTM*MTS),1/MTS]	Median
	(0,1/(FTM*MTS))	Faster
Median	(MTS,+∞)	Straggler
	[1/FTM,MTS]	Median
	(0,1/FTM)	Faster
Faster	(FTM*MTS,+∞)	Straggler
	[FTM,FTM*MTS]	Median
	(0,FTM)	Faster

3.3 Computational Capacity Determination

After predicting the state of each node according to the input data characteristics, we next estimate computational capacity of each node before we can conduct pre-scheduling.

The computational capacity of a node refers to the amount of data that the node can process in one batch. Considering the periodicity of the recurring micro-batch jobs, we estimate the

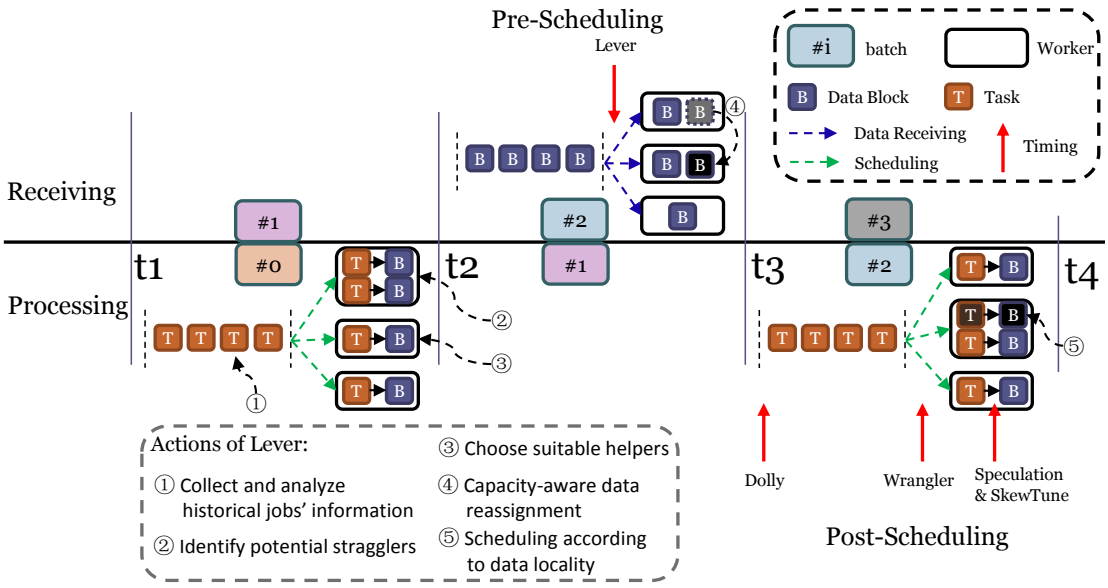


Fig. 6: Different scheduling timing of Lever and post-scheduling methods

computational capacity based on the Iterative Learning Control (ILC [26]) model, which is designed to perform tracking control of the systems working in a repetitive mode. Repetition allows the system to improve the tracking accuracy through iterations. This learning process uses information from previous repetitions to improve the estimation and can achieve more accurate results iteratively. This scenario is similar to the batched stream processing jobs.

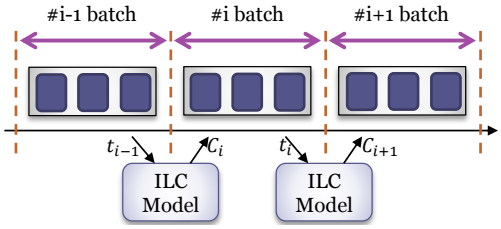


Fig. 8: The principle of evaluating computational capacity with ILC.

Figure 8 shows the principle of the ILC algorithm. Our objective is to continuously approximate the real computational capacity. The task finish time of each node and computational capacity in the previous batch are passed to the ILC model as the learning parameters. The ILC model is used to estimate the computational capacity of the processing nodes for the next batch. These actions repeat from one batch to the next. A model of estimating the capacity of a node is of the following form:

$$C_{i+1} = C_i + K * \Delta t \tag{1}$$

$$\Delta t = t_{ideal} - t_i \tag{2}$$

where C_i is the capacity during the ith batch, Δt is the deviation between the node's finish time and the ideal finish time during the ith batch. K is a design parameter representing the operations on Δt. In Lever, K is set to C_i/t_i. t_{ideal} is the ideal finish time of the node in the ith batch and can be obtained by computing the median node's finish time. We repeat this process for every batch.

3.4 Capacity-Aware Data Reassignment

After grouping the nodes into three groups, the nodes in the faster group are eligible to act as helpers to fetch part of the stragglers' workload. Nodes in the straggler group are helpees to whom helpers should provide assistance. After we figure out node's type(helper or helpee) along with each node's capacity, we can conduct capacity-aware data reassignment.

3.4.1 *Choosing Helpers*

First, Lever selects some nodes from faster group as candidate helpers. Considering that candidate helpers will afford extra work from stragglers, these helpers should have large computational capacity in order to avoid the selected helpers being stragglers in next batch again. So, Lever sorts the fasters' list according to their capacities by descending order and selects head r fasters to be helpers. Parameter r are tune-able. In our experiments, when r is set to 4 or higher, near-ideal performance is achieved.

3.4.2 *Data Reassignment*

Theoretical Data Assignment. In the ideal case, all the tasks should be completed simultaneously. So, the system should increase the amount of load in the faster nodes and reduce those in the stragglers to minimize the makespan. Assume that there are n nodes. Let L_i and C_i denote the input load and the computational capacity of the ith node respectively. Let L_i' denote the input load under Lever's pre-scheduling plan. Let t_i denotes the finish time of ith node. We have:

$$t_i = L_i'/C_i \tag{3}$$

$$\sum_{i=1}^{n} L_i = \sum_{i=1}^{n} L_i' \tag{4}$$

In the ideal case, our optimization goal is $\delta^2 = D(t_i) = 0$. So, we have $t_1 = t_2 = ... = t_n$. Then, we can get:

$$\frac{L_1'}{C_1} = \frac{L_2'}{C_2} = ... = \frac{L_n'}{C_n} \tag{5}$$

Considering equation 4, we can derive:

$$\frac{L_1\prime}{C_1} = \frac{L_2\prime}{C_2} = ... = \frac{L_n\prime}{C_n} = \frac{\sum_{i=1}^{n} L_i\prime}{\sum_{i=1}^{n} C_i} = \frac{\sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} C_i} = \frac{L_i\prime}{C_i} \tag{6}$$

The load L_i' can be denoted as:

$$L_i\prime = \frac{\sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} C_i} * C_i \tag{7}$$

So, the load we need to migrate to or from can be denoted as:

$$\Delta L = L_i\prime - L_i = \frac{\sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} C_i} * C_i - L_i \tag{8}$$

Capacity-Aware Reassignment Strategy. After we select the candidate helpers, we can conduct capacity-aware data reassignment according to each node's capacity. Figure 9 shows the principle of capacity-aware data reassignment strategy. Assume that we have n helpers, the node i is characterized by the vector (L_i,C_i). For each straggler, characterized by (L_s,C_s), its input data is assigned to all the selected helpers according to their capacity and load. According to Equation 8, the load share L_{sToj} which is dispatched to the jth helper can be denoted as:

$$L_{sToj} = \frac{L_s + \sum_{i=1}^{n} L_i}{C_s + \sum_{i=1}^{n} C_i} * C_j - L_j \tag{9}$$

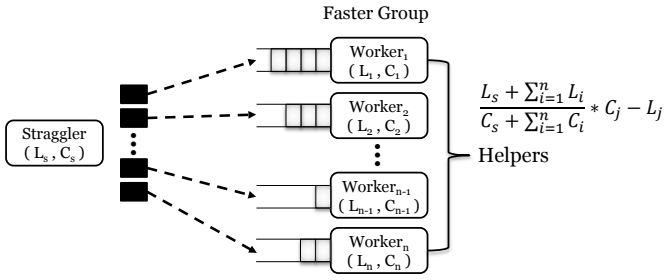


Fig. 9: Capacity-aware reassignment strategy.

Adapting Block Size. Lever adapts block size according to capacity-aware data reassignment strategy. Batched stream processing system aggregates input records into *blocks*. Several *blocks* together make up a *batch*. These *blocks* then are scheduled and distributed to executors for processing as so called *tasks*. In order to prevent large task being straggler task in straggler node, Lever splits large task' block into several small sub-blocks and then migrates some of them to helper nodes.

Figure 10 shows how Lever adjusts block size in batched stream processing system. First, a *receiver* receives input stream records and stores them into an array buffer in *BlockGenerator*. Then, Lever splits the array buffer according to a split ratio which has been decided by reassignment strategy. These array buffer slices generate blocks and are pushed into block queue. After that, every block will be dispatched to a node which has the corresponding capacity with split ratio. Pre-schedule procedure is done. When scheduling tasks, executors fetch local blocks firstly by data locality for processing. The complete pseudo code of the pre-scheduling algorithm is presented in Algorithm 1. Lines 1∼15 describes potential straggler identification. Lever transforms node's state according to initial state, FTM and MTS (lines 9∼13). Lines 17∼23 describes how Lever leverage ILC model to evaluate node capacity. Lines 27∼37 states the detailed procedure of

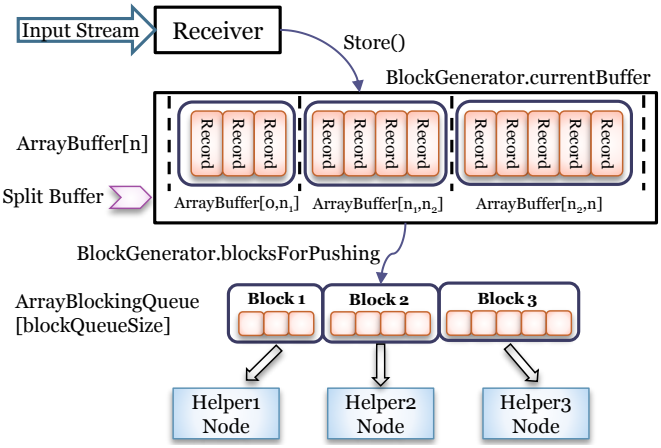


Fig. 10: Adapt block size by splitting block array buffer.

capacity-aware data reassignment. How much load should we pre-schedule from straggler to faster node is decided in lines 33∼35.

4 System Implementation

We have implemented Lever based on Spark Streaming [27], a popular open-source distributed batched stream processing system, which is an extension of the cluster computing framework Apache Spark [28]. We choose Spark Streaming because it is a typical batched stream processing system based on Spark, a fast and general engine for large-scale data processing which powers a stack of libraries including SQL, machine learning, graph processing and stream processing. Spark Streaming has been widely adopted in both academia and industry, and has also been deployed on production clusters of many corporations [9]. In this section, we describe the detail of our system implementation. The source code of our system can be found at https://spark-packages.org/package/trueyao/spark-lever.

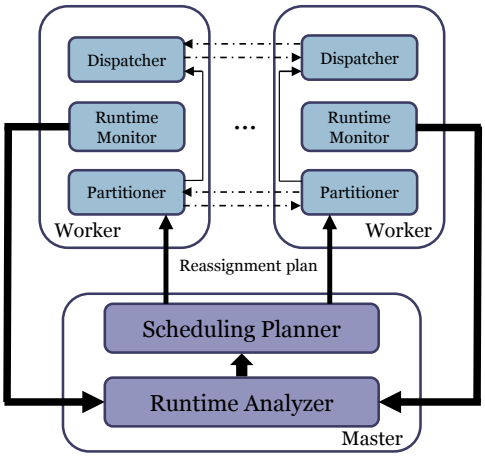


Fig. 11: Implementation of Lever

Figure 11 provides an overview of the implementation of Lever. We add two components (i.e., *Runtime Analyzer* and *Scheduling Planner*) to the master, and three components (i.e., *Runtime Monitor*, *Partitioner* and *Dispatcher*) to the worker, respectively. *Runtime Monitor* located on each worker periodically detects the worker's runtime information such as load

Algorithm 1 Pre-Scheduling Algorithm

Input: Historical job profiles in previous batches
Output: Scheduling decisions

Procedure Identify Potential Stragglers
Get the previous batches' job execution information
Sort the nodes descending by their finish time
for each node N **do**
  Intialize(node N's state)
end for
Compute the input load's gradient:
$inputRatio = \frac{newInputRatio}{oldInputRatio}$
Compute transition condition FTM and MTS
for each group G **do**
  for each node N in group G **do**
    transform(node N's initial state, FTM, MTS)
  end for
end for
Output final stragglers and fasters
End Procedure
Procedure Evaluate Computational Capacity
for each node **do**
  Initial condition: $C_1 = Throughput$
  In ith batch, compute current batch's deviation of node's finish time:
  $t_{ideal} = t_{median(i)}, \Delta t = t_{ideal} - t_i$
  Compute (i+1)st batch's capacity:
  $C_{i+1} = C_i + \frac{C_i}{t_i} * \Delta t$
  Assign corresponding load when receiving (i+1)st batch's data
  In (i+1)st batch, compute current batch's deviation of node's finish time:
  $t_{ideal} = t_{median(i+1)}, \Delta t = t_{ideal} - t_{i+1}$
  compute the next batch's capacity:
  $C_{i+2} = C_{i+1} + \frac{C_{i+1}}{t_{i+1}} * \Delta t$
end for
End Procedure
Procedure Capacity-Aware Data Reassignment
Sort faster nodes according to their capacities by descending order
Choose head *n* the nodes as helpers
Compute the sum of helpers' capacity:
$sumOfCapa = \sum_{i=1}^{n} C_i$
Compute the sum of helpers' load:
$sumOfLoad = \sum_{i=1}^{n} L_i$
for each straggler node **do**
  for each helper node **do**
    Compute the allocated load share(split ratio):
    $L_{stoi} = \frac{L_s + sumOfLoad}{C_s + sumOfCapa} \times C_i - L_i$
    Update each node's (C_i, L_i):
    $C_i' = C_i$, $L_i' = L_i + L_{stoi}$
  end for
  Invoke an interface *splitBlockBuffer* to split block array buffer according to split ratio
  Invoke *blockTransferService* to migrate the data block to corresponding nodes
end for
End Procedure

and processing speed, and then reports to *Runtime Analyzer*. By analyzing such information together with the jobs' detailed execution information, *Runtime Analyzer* identifies the stragglers and evaluates each node's computational capacity. These results will be passed to *Scheduling Planner* to make pre-scheduling decisions. *Scheduling Planner* makes a reassignment plan about how to partition and pre-schedule stragglers' work to other nodes. *Partitioner* is responsible for partitioning stragglers' excess work according to the specified proportion derived from *Scheduling Planner*. *Dispatcher* distributes every piece of the partitioned work in a straggler to the selected helpers. Further details of the main components to implement Lever on Spark Streaming are described below.

Runtime Monitor. A *Runtime Monitor* is located on each worker and periodically detects the worker's information. In order to periodically detect the worker's runtime information, we introduce several data collection functions in the *executor* to collect the needed information such as task finish time, and input data size. We also add an accumulator in the *worker*. Once a task finishes, its corresponding information is encapsulated into a message and then sent to the accumulator, which gathers these messages and reports to *Runtime Analyzer* through an asynchronous RPC based on Akka.

Runtime Analyzer. Once a batch has finished, *Runtime Analyzer* begins to analyze the statistics information from *Runtime Monitor*. We create a new component which maintains a table (implemented using *HashMap*) in the *master*. This table is used for recording and updating the workers' information.

Scheduling Planner. This component receives messages from *Runtime Analyzer*. It mainly includes a function responsible for running our capacity-aware pre-scheduling algorithm and an output table which records the pre-scheduling data assignment plan. We also design callback functions in *JobScheduler* and *TaskScheduler* in order to get scheduling information feedback and adjust the task scheduling accordingly.

Partitioner. We modify *BlockGenerator* and *ReceiverSupervisorImpl* which are basic modules of Spark Streaming to implement the Partitioner. We implement a partition function *splitBlockBuffer* to divide the receiving buffer into the specified data shares. These fragments are delivered to *BlockManager* as *block*.

Dispatcher. This component carries out two tasks: (1) getting the name of the host that a data block will be assigned to according to the assignment plan, and (2) invoking the *blockTransferService* to transfer the data block. We modify the system call *uploadBlockSync* in Lever to implement the remote block assignment.

5 Experimental Evaluation

In this section, we evaluate Lever's performance under various workloads. First, we describe the experimental environment, and then present the experimental results.

5.1 Experimental Setup and Workloads

Experimental Setup. Our evaluations are conducted on a heterogeneous cluster consisting of two types of machines with different configurations. The first kind of machines is comprised of two dual-core Intel Xeon E5-2670 2.6 GHz CPUs, 16GB memory, and 210GB disks. The second kind of machines consists of two dual-core 1.2 GHz Intel CPU, 4GB memory, and 210GB disks. All computers are interconnected with 1Gbps Ethernet cards. We use spark-1.3.0 as the distributed computing platform and Redhat Enterprise Linux 6.2 with the kernel version 2.6.32 as the OS. In multi-tenant environment, we use hadoop version 2.6.0 (a.k.a., YARN) as the resource management platform. In standalone mode, only one executor is launched in each machine. So, Spark executor can use all the resources of each node. In YARN mode, considering that resources need to be shared with other applications and jobs, we allocate 2 cores and 1GB memory to Hadoop containers.

Workloads. Table 2 describes the benchmarks used in our experiments. Similar to previous work [8], we choose three typical

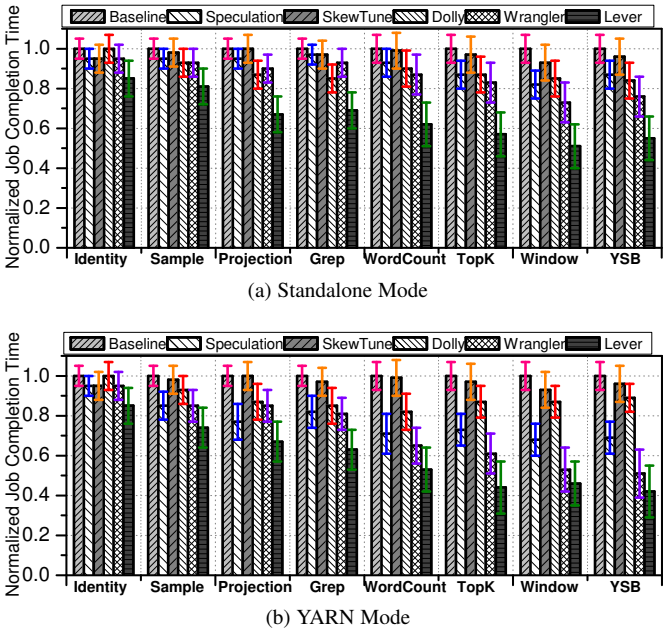
TABLE 2: Benchmarks

Benchmark	Description	Complexity	Resource Preference	Source
Identity	Simply reads input and takes no operations on it	Single Step	None	HiBench [29]
Sample	Samples the input stream according to specified probability	Single Step	None	HiBench [29]
Projection	Extracts a certain field of the input	Single Step	Network Intensive	HiBench [29]
Grep	Checks if the input contains certain strings	Single Step	Network Intensive	DStream [8]
WordCount	counts the number of each word in input text	Multi Step	CPU Intensive	DStream [8]
Topk	Finds the k most frequent words over the specified window	Multi Step	CPU Intensive	DStream [8]
Window	Counts the number of each word in a sliding window	Multi Batch	CPU Intensive	-
YSB	Advertising using spark streaming in Yahoo streaming benchmark	Multi Step	CPU Intensive	YSB [30]

applications i.e. Grep, WordCount, and TopK. In addition, we also use the HiBench [29] benchmark including Identity, Sample and Projection. We rewrite the HiBench benchmark based on receiver model. Windowed WordCount (Window) and Yahoo streaming benchmark(YSB) [30] for spark are also used to evaluate Lever's performance.

Other configurations. In all our experiments, we use the Spark default data locality wait time (3s), therefore Delay Scheduling [31] does not interfere with our experimental results. For Speculative Execution, we set the speculation interval, speculation quantile and speculation multiplier as 100ms, 0.5, 1 respectively. These settings let Speculative Execution achieve the best performance in our environment based on our tuning results. We implemented Skewtune, Dolly and Wrangler's prototypes. According to [15] [20] [21], we make the following settings. We adopt local scan for Skewtune and set the wait duration of delay assignment ω as 200ms in Dolly. For Wrangler, we set the confidence measure ρ as 0.7 and set the interval Δ, which decides the frequency of node resource usage counters are collected, as one batch.

Unless specified otherwise, we set batch interval as 2 seconds and input records as 100 bytes. For windowed wordcount, we set window duration as 10 seconds and slide duration as 2 seconds. All input data blocks only have one replica. We do not fetch the final results until the results are stable after each benchmark runs about 10 minutes. Each experiment is repeated five times and we present the median numbers. The baseline is original spark streaming with speculation closed.



(a) Standalone Mode

(b) YARN Mode

Fig. 12: Normalized job completion time in two deploying modes

5.2 Job Completion Time

We first test the improvements in job completion time using Lever in two usage mode, i.e. standalone mode and YARN mode. Figure 12 reports the normalized job completion time when compared with other strategies in two heterogeneous environments. Lever can significantly improve performance when running Projection, Grep, WordCount, Topk, Windowed WordCount and YSB. But Lever improves a little for Identity and Sample relatively. This is because the performance improvement depends on the type of workloads. Identity and Sample are of simple execution logic. Taking Identity for example, it just simply reads input data and takes no operations on them, and then outputs these raw data. Therefore, stragglers have little impact on these workloads.

First, we present the results in standalone mode. Compared to baseline, Lever improves average job completion time by 32.31%, 30.72%, 37.54%, 42.19%, 48.17% and 44.64% for Projection, Grep, WordCount, Topk, Windowed WordCount and YSB respectively. Speculation and Skewtune work much worse in our experiments because they need to spend a long time on detecting stragglers and data skew. Lever pre-schedules input data before task scheduling and avoids the detecting and migrating overhead. As a result, Lever outperforms other strategies for the performance of Projection (by up to 29.47%, 32.06%, 22.99% and 25.56% compared to Speculation, Skewtune, Dolly and Wrangler), Grep (by up to 28.87%, 29.11%, 18.82% and 25.81% compared to Speculation, Skewtune, Dolly and Wrangler), WordCount (by up to 33.33%, 37.37%, 31.11% and 28.74% compared to Speculation, Skewtune, Dolly and Wrangler), Topk (by up to 34.48%, 41.24%, 34.48% and 31.33% compared to Speculation, Skewtune, Dolly and Wrangler), Windowed WordCount (by up to 37.81%, 45.16%, 40.11% and 30.14% compared to Speculation, Skewtune, Dolly and Wrangler) and YSB (by up to 36.78%, 42.71%, 34.52% and 27.63% compared to Speculation, Skewtune, Dolly and Wrangler) in standalone mode.

Second, we present the results in YARN mode. We use Spark Streaming running wordcount as the background job. Compared to the baseline, Lever improves average job completion time by 32.17%, 36.44%, 46.91%, 55.21%, 53.36% and 57.28% for Projection, Grep, WordCount, Topk, Windowed WordCount and YSB respectively. Speculation and Wrangler behaves much better

than Skewtune and Dolly in YARN mode. Speculation detects stragglers by being aware of task progress. The task progress is affected by resource contention. Wrangler has a machine learning model to detect stragglers by collecting resource utilization metrics of a node. That is to say, Speculation and Wrangler all can be aware of the resource fluctuation. However, Skewtune can only detect data skew and can not be aware of resource contention. Dolly has not the ability to adapt to resource changes and it clones tasks and increases 2x work. So Speculation and Wrangler are more suitable for dealing with stragglers in resource-sharing environments. Also Speculation and Wrangler can achieve better performance in YARN mode than in standalone mode. As a result, Lever outperforms other strategies for the performance of Projection(by up to 12.99%, 33.24%, 22.99% and 21.18% compared to Speculation, Skewtune, Dolly, Wrangler), Grep(by up to 23.17%, 35.15%, 25.88% and 22.22% compared to Speculation, Skewtune, Dolly and Wrangler), WordCount(by up to 25.35%, 46.52%, 35.37% and 18.64% compared to Speculation, Skewtune, Dolly and Wrangler), Topk(by up to 39.73%, 54.64%, 49.43% and 27.87% compared to Speculation, Skewtune, Dolly and Wrangler), Windowed WordCount(by up to 32.35%, 50.54%, 47.13% and 13.21% compared to Speculation, Skewtune, Dolly and Wrangler) and YSB(by up to 39.13%, 56.25%, 52.81% and 17.65% compared to Speculation, Skewtune, Dolly and Wrangler) in YARN mode.

In summary, Lever behaves better in YARN mode than in standalone mode. It is because Lever can adjust data reassignment quickly by analyzing recurring jobs' historical information. For example, if resource competition is very severe on one node in this batch, Lever is able to quickly detect resource fluctuation and make proper pre-scheduling decisions next batch, thus avoiding straggler tasks.

5.3 Stage MakeSpan

In order to understand how Lever improves the performance, in this test we show a detailed analysis of stage makespan for every executor. The stage makespan of a executor is defined as the time from when the first task starts to when the last task finishes in the executor.

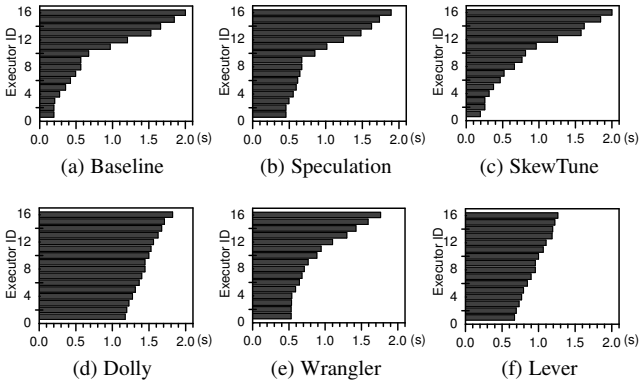


Fig. 13: Stage makespan in executors on Standalone Mode

Figure 13 presents the results when running WordCount on Standalone mode in batch interval of 2 seconds. Makespan is decided by the last task's completion time. As we can see, load unbalancing under Speculation, Skewtune and Wrangler are quite remarkable. They need to spend much time on detection and migration thus leading to lots of idle time in some executors. For Speculation, Executor 1 is free for 1.436s of 1.891s' JCT (Job Completion Time). For Skewtune, Executor 1 is idle for 1.806s of 1.997s' JCT. For Wrangler, Executor 1 has nothing to do for 1.231s of 1.762s' JCT. They can not response to stragglers quickly and migrate them to free fast executors in such a short batch interval. Although Dolly wastes less time and balances load very evenly, it clones tasks, which means that it needs to complete 2x amount of work. In the ideal case, all the nodes should complete their tasks at almost the same time. Lever behaviors much better because it pre-schedules data and adjusts task size when receiving data. So when tasks are running, all nodes can progress at almost the same rate. For Lever, Executor 1 is idle for 0.592s of 1.263s' JCT. Figure 14 shows the results when running WordCount on YARN mode.

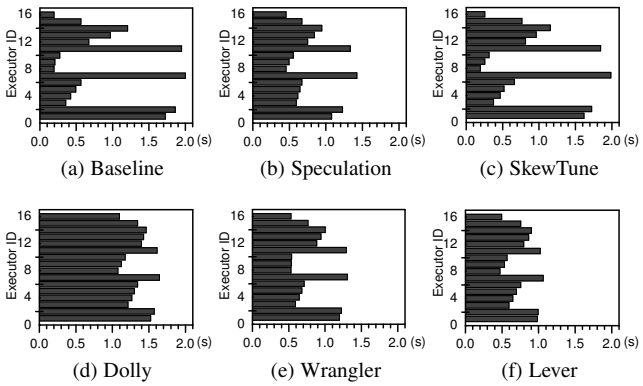


Fig. 14: Stage makespan in executors on YARN Mode

In summary, Lever can achieve much better load balancing through capacity-aware data pre-scheduling, thus avoiding detection, migration and cloning. As a result, Lever can minimize makespan and improve the performance significantly.

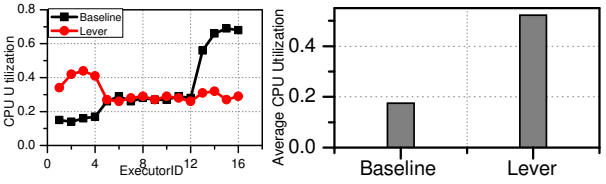
5.4 CPU Utilization

In this section, we try to explain Lever's improvements from a perspective of CPU utilization. We test each executor's CPU utilization and average CPU utilization. Figure 15 presents the results when running WordCount on Standalone mode. We measure each executor's CPU utilization by reading the load average of 15 minutes printed by *top* command. We measure average CPU utilization as follows. We sum up all tasks' processing time called total task time, i.e. CPU execution time. The total CPU time can be computed as the number of cores multiply by job completion time (JCT). The average CPU utilization can be expressed as:

$$AverageCPUUtilization = \frac{TotalTaskTime}{Cores * JCT} \tag{10}$$

In batched stream systems, the average CPU utilization is higher than average executor CPU utilization because there exists free time between job completion time and batch interval. For brevity, we show the results when running WordCount in baseline. Other benchmarks have similar results.

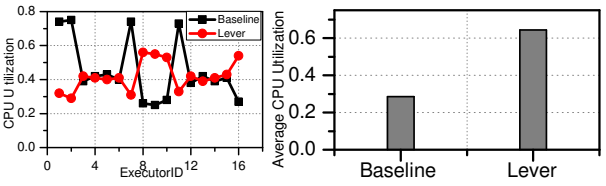
Figure 15(a) shows each executor's CPU utilization. From this figure, we can see that Executor 13, 14, 15, 16 have high CPU utilization up to 69.53% in Baseline. Obviously, these executors are of heavy load. Actually, these executors are launched on slow



(a) Executor CPU Utilization (b) Average CPU Utilization

Fig. 15: CPU Utilization on Standalone mode. Executor CPU utilization reflects whether this executor is overloaded or underloaded. Average CPU utilization reveals whether CPU resources are being utilized efficiently in one batch.

nodes. Executor 1, 2, 3, 4 have low CPU utilization up to 14.61% in Baseline. These executors are deployed on fast nodes. Their CPU resources can not be fully utilized. However, in Lever, load of Executor 13, 14, 15, 16 can be migrated to Executor 1, 2, 3, 4 efficiently, leading to Executor 13, 14, 15, 16 with 27.37% CPU utilization and Executor 1, 2, 3, 4 with 52.12% CPU utilization. The average CPU utilization is 17.51% and 52.27% for Baseline and Lever respectively. This is not only due to the reduction in job completion time, but also because Lever reduces the total task execution time. Figure 16 presents the results when running WordCount on YARN mode.



(a) Executor CPU Utilization (b) Average CPU Utilization

Fig. 16: CPU Utilization on YARN mode

In summary, Lever can decrease the CPU load of slow nodes and improve CPU utilization of fast node. Lever can also improve average CPU utilization from 17.52% to 52.23%.

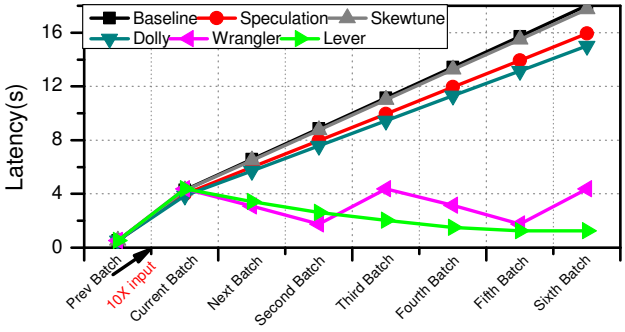
5.5 Adaptability for burst load

In this testing scenario, we give one node burst load to evaluate the robustness and the convergence time of Lever. The burst load results in that task completion time exceeds batch interval. We also test other techniques' effectiveness under this situation.

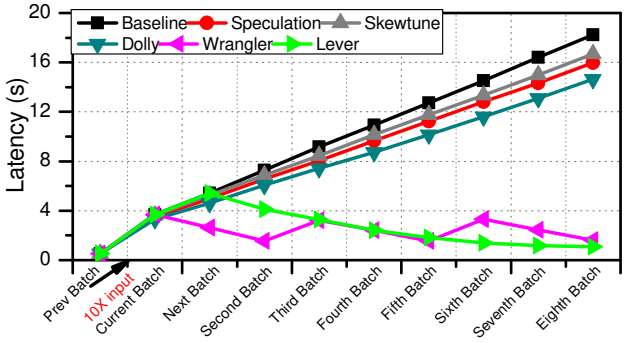
From the test result in Figure 17, we observe that both Lever and Wrangler can achieve much better performance. Baseline, Speculation, Skewtune and Dolly can not process subsequent jobs timely because their cloning and hysteresis reaction are not enough, leading to a large number of subsequent jobs are queueing in scheduler's waiting queue. The scheduling delay will increases monotonously.

Although Wrangler can continue to process subsequent jobs, it is not stable enough with delay jitter. This is because Wrangler only cares about stragglers in one batch. The nodes of light load (actually are potential stragglers) in current batch will be scheduled many tasks in the next batch, leading to these nodes being overloaded.

Lever avoids this problem by analyzing recurring jobs' load fluctuation in previous batches. As shown in Figure 17 (a), when



(a) Latency when running WordCount under burst load on a straggler node



(b) Latency when running WordCount under burst load on a faster node

Fig. 17: Adaptability for burst load

giving burst load to a straggler node, Lever can converge to an ideal latency in six batches. However, when giving burst load to a faster node, Lever is late on identifying state transition of the faster node for about one batch, as shown in Figure 17 (b). Only when a faster node shows the characteristics of a straggler node in one batch, can Lever adapts its scheduling decisions in the next batch. Lever converges to an ideal latency in seven batches when giving burst load to a faster node.

In summary, Lever can adapt to burst load effectively and converge to a steady state quickly.

5.6 Data Locality and Queued Tasks

In this test, we conduct a statistical analysis about data locality and queued tasks whose waiting time is greater than three quarters of batch interval. We observe that in our experiments if a task waits for three quarters of batch interval, it means that this node is overloaded and the batch processing time will exceed the batch interval.

As shown in Figure 18, for Baseline, although most of the tasks can execute in the local node, there are 33.5%, 39.7%, 44.9% tasks waiting for more than three quarters of batch interval for Grep, WordCount and Topk, respectively. These tasks are from straggler nodes. In order to illustrate the difference, we take WordCount for example. Dolly and Wrangler only have up to 23.1% and 17.3% tasks waiting for more than three quarters of batch interval. However, they only have 71.2% and 81.5% node local tasks. They need to migrate tasks to remote nodes at runtime. This migration causes extra overhead and neutralizes benefits from reduction in the number of waiting tasks. Speculation only detects and migrates a small number of tasks. Skewtune performs much worse because it spends so long time on detection that it can not react to straggler and data skew quickly.

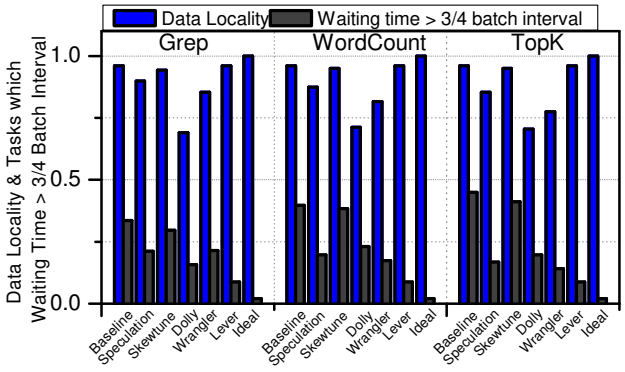


Fig. 18: Data Locality and Tasks whose waiting time is greater than 3/4 batch interval when running Grep, WordCount and Topk

In the ideal case, all nodes execute local tasks and there are not tasks queued for more than three quarters of batch interval. Lever can achieve 96% data locality and 8.7% waiting tasks. The reason why Lever can not reach the ideal state is that Lever relies on the estimation of node's capacity and ignores the median nodes. This impedes Lever's perfect load balancing.

In summary, compared to other strategies, Lever can achieve better data locality and reduce the number of tasks waiting for long time.

5.7 Deconstructing Performance

In this section, we deconstruct Lever and test each component's performance to have a deep understanding about Lever's improvements. We deconstruct Lever into three parts, i.e. identify potential stragglers based on recurring jobs (IPS), computational capacity determination using Iterative Learning Control (ILC) and capacity-aware data reassignment (CADR). We fetch 1800 batches' information of WordCount to make a statistical analysis about job completion time. Figure 19 presents the results.

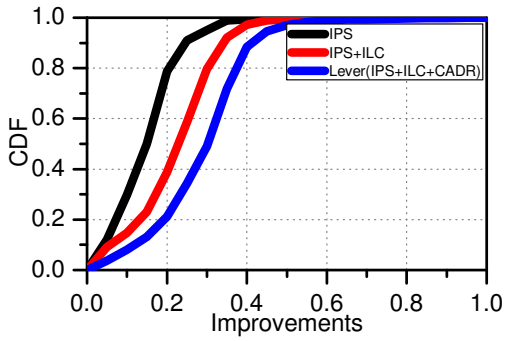
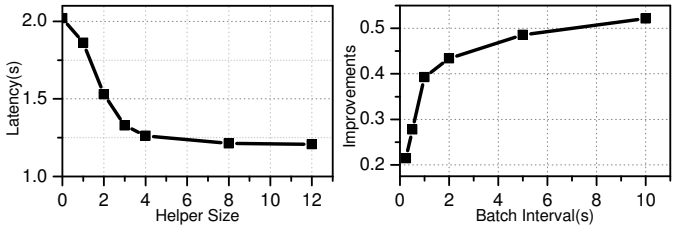


Fig. 19: Lever improvements. We deconstruct Lever into three parts, i.e. identify potential stragglers based on recurring jobs (IPS), computational capacity determination using Iterative Learning Control (ILC) and capacity-aware data reassignment (CADR).

As we can see, more than 62% batches can achieve 10%∼25% improvements using IPS. 98% batches' improvements is below 35%. By using IPS + ILC, more than 69% batches can achieve 15%∼35% improvements. 98% batches' improvements is below 45%. Under IPS + ILC + CADR, more than 73% batches have 20%∼45% improvements. 98% batches' improvements is below 60%.

5.8 Sensitivity Study

This section reports on tests used to determine good settings for Lever parameters. We vary helper size and batch interval to test Lever's performance. For brevity, we show sensitivity results when running WordCount, although similar results hold for other benchmarks.



(a) Helper Size (b) Batch Interval

Fig. 20: Sensitivity Analysis. We vary the size of helper group and batch interval to evaluate Lever's performance.

Helper Size Test. Recall that the helper group is the set of workers to whom a worker is eligible to provide assistance. Figure 20(a) shows the results of varying the helper size from zero helpers, which is running in Baseline mode, to twelve helpers for Lever. The results show that, once the helper size is set to 4 or higher, near-ideal performance is achieved. Closer inspection reveals that using only six helpers provides the best performance. Of course, the more the helpers are, the better performance Lever can achieve. Because Lever can distribute stragglers' work more evenly among more helpers. But, the difference between settings from 4 to 12 is negligible.

Batch Interval Test. The batch interval (also called as batch size) of a batched stream system significantly affects the performance and also impacts Lever's improvements. Figure 20(b) shows the results of varying the batch interval from 0.25s to 10s. Considering that batched stream systems are usually deployed in a real-time or near real-time scenario, we do not use larger batch interval. As shown in this figure, we can see that when batch interval is smaller than 1s, Lever's effect is not evident. When batch interval is 0.5s, Lever can achieve the improvements by 27.78%. When batch interval comes to 0.25s, Lever can only achieve 21.47%'s improvements. That is because when using very small batch interval, there is a very small probability producing stragglers. Stragglers wouldn't fall behind significantly. As batch interval increases, the probability becomes higher. The impact of stragglers is also getting more and more serious. For example, for a 2s' batch interval, most of the tasks are normal tasks completed in 0.5s. However, if one task proceeds more than 1s, it could be marked as straggler task. So when batch interval is 2s, Lever's improvements can arrive at 43.39%.

In summary, choosing 4 helpers in our experimental environments is enough for Lever to achieve good performance. The batch interval should not be too small. If the batch interval is less than 1s, Lever may behaves worse.

6 Discussion and Future Work

Alternative identifying stragglers. Lever uses a lightweight method to predict stragglers. There is no doubt that using other methods such as machine learning algorithms might be able to get a more accurate result. However, considering the low latency

of batched stream processing, the overhead of pre-scheduling should be as small as possible. We do not adopt more complicated solutions because it is enough to identify stragglers in batched stream processing.

Compatibility with post-scheduling. Note that pre-scheduling and post-scheduling are not mutual exclusive. They are actually compatible with each other and can be integrated into a system. If pre-scheduling misses some stragglers, then we can use post-scheduling approaches to tackle them. How to effectively coordinate pre-scheduling with post-scheduling will be a valuable problem in our future work.

Multiple-stage and shuffle-heavy jobs. Lever has its limitations because it mainly affects the stragglers in the first stage of a job. In our experiments, the representative benchmarks are all two-stages. According to our observations, the tasks' execution time in the first stage of a job accounts for a large proportion of the job completion time. So, Lever performs much better in these scenarios. However, when running multi-stage jobs or shuffle-heavy jobs, the bottlenecks will be moved to shuffle operations or other aspects. We plan to consider the influence of multi-stages or shuffle-heavy tasks and how to pre-schedule data at shuffle stage in the future. It will extend the usage of Lever.

7 Related Work

Our work is related to the research in batched stream processing and straggler mitigation on heterogeneous clusters. We discuss the most related work in this section.

Batched Stream Processing and Incremental Data Processing Systems. Batched stream processing systems collect received data into batches and periodically process them using MapReduce-style batch computations. The typical systems include Comet [6] which is structured on DryadLINQ, HOP [5] which leverages the power of batch framework MapReduce, and Spark Streaming [8] which is built on top of Spark. These systems take full advantage of characteristics in batch processing engine such as high throughput and fault-tolerance. Some other systems [32] [33] intent to modify batch framework to adapt to the requirements of stream processing. Other stream processing systems such as Borealis [34], TimeStream [35], Naiad [11], Storm [36] and Heron [37] are based on the *continuous operator model*. In this model, the streaming computation is expressed as a graph of long-lived operators that exchange messages with each other to process the streaming data. Incremental bulk processing systems such CBP [38], Percolator [39] and Incoop [40] allow updated view of processed data set to be maintained by incrementally and efficiently recomputing the updates to the input data. In such systems, the recomputation is triggered whenever an update to the input dataset is detected. A-scheduler [41] proposes an adaptive scheduling approach for Spark Streaming that dynamically schedules multiple jobs concurrently using different policies based on their data dependencies and automatically adjusts the level of job parallelism and resource shares among jobs based on workload properties. [42] designs an online adaptive algorithm based on Fixed-Point Iteration for batched stream processing system. By using job statistics of prior batches, it is able to quickly adapt the batch size under changes in data rates, workload behaviors and available resources. Lever is also inspired by this work. Drizzle [12] observes that while streaming workloads require millisecond-level processing, workload and cluster properties change less frequently. It decouples the processing interval from the coordination interval used for fault tolerance and adaptability and implements a prototype based on Spark Streaming.

Straggler and Data Skew on Heterogeneous Clusters. Load imbalance on heterogeneous clusters are common phenomenon in cluster computing environments because of heterogeneity, straggler, data skew and so on. Many techniques have been presented to solve these problems. The typical straggler mitigation approaches are Speculative Execution [3] [43], Mantri [17], Dolly [20], GRASS [44] and Wrangler [21]. Speculative Execution marks slow tasks as stragglers and launches a redundant copy of a task-in-progress on a different node. Using real-time progress reports, Mantri monitors, detects and acts on outliers by restarting outliers, network-aware placement of tasks and protecting outputs of valuable tasks. Dolly is a replication-based method, and it proposes full cloning of small jobs by delay assignment. Dolly do not need to wait to observe before acting with a proactive approach of cloning jobs. But it incurs extra resources. GRASS is designed for approximation jobs, and it delicately balances immediacy of improving the approximation goal with the long term implications of using extra resources for speculation. Wrangler automatically learns to predict stragglers using a statistical learning technique based on cluster resource utilization counters. It is a straggler-avoid method. The typical skew mitigation techniques are Scarlett [45], SkewTune [15]. Scarlett replicates blocks based on their popularity by accurately predicting file popularity and working within hard bounds on additional storage. SkewTune solves the problem of load balancing in MapReduce-like systems by identifying the task with the greatest expected remaining processing time and redistributing the unprocessed data from the stragglers to other workers. Sparrow [46] is a decentralized scheduler designed for scheduling millions of tasks per second while offering millisecond-level latency and high availability. It isn't developed for solving straggler problems. [47] proposes a cross-platform resource scheduling middleware which aims to improve the resource utilization and application performance in multi-tenant Spark-on-YARN clusters.

8 Conclusion

Optimizing batched stream processing system in heterogeneous environments has been a challenging problem. This paper presents Lever, a pre-scheduling straggler mitigation framework that exploits the predictability of recurring batched stream jobs to optimize the assignment of data. Lever identifies potential stragglers by analyzing execution information of historical jobs and introduces Iterative Learning Control (ILC) model to evaluate nodes' capacity. Furthermore, Lever carefully chooses helpers and optimizes the reassignment of job input data according to each node's capacity proportion. This feedback-control loop makes Lever achieve much better load balancing and avoid high execution overhead. Lever has been implemented on the top of Spark Streaming. It is also open source and has been included in Spark Packages at https://spark-packages.org/package/trueyao/spark-lever. We conduct various experiments to validate the effectiveness of Lever. Experimental results demonstrate that Lever reduces job completion time by 30.72% to 42.19% and outperforms traditional techniques significantly.

References

[1] M. Ni Lionel, J. Xiao, and H. Tan, "The golden age for popularizing big data," *Science China Information Sciences*, vol. 59, no. 10, pp. 235–237, 2016.

[2] D. Li, J. Cao, and Y. Yao, "Big data in smart cities," *Science China Information Sciences*, vol. 58, no. 10, pp. 1–12, 2015.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *in Proc. of OSDI'04*, pp. 137–149, 2004.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *in Proc. of HotCloud'10*, pp. 10–16, 2010.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," *in Proc. of NSDI'10*, pp. 21–35, 2010.

[6] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," *in Proc. of SoCC'10*, pp. 63–74, 2010.

[7] "Hstreaming," http://www.hstreaming.com.

[8] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," *in Proc. of SOSP'13*, pp. 423–438, 2013.

[9] "Spark summit," http://spark-summit.org.

[10] "Storm," http://storm.apache.org.

[11] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," *in Proc. of SOSP '13*, pp. 439–455, 2013.

[12] S. Venkataraman, A. Panda, K. Ousterhout, and et al., "Drizzle: Fast and Adaptable Stream Processing at Scale," *in Proc. of SOSP'17*, pp. 374–389, 2017.

[13] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. a. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," *in Proc. of SoCC '12*, pp. 1–13, 2012.

[14] D. Cheng, J. Rao, Y. Guo, C. Jiang, and X. Zhou, "Improving performance of heterogeneous mapreduce clusters with adaptive task tuning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 774–786, 2017.

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating Skew in Mapreduce Applications," *in Proc. of SIGMOD'12*, pp. 25–36, 2012.

[16] Q. Chen, J. Yao, and Z. Xiao, "Libra: Lightweight data skew mitigation in mapreduce," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2520–2533, 2015.

[17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-reduce Clusters Using Mantri," *in Proc. of OSDI'10*, pp. 24–37, 2010.

[18] D. Cheng, P. Lama, C. Jiang, and X. Zhou, "Towards Energy Efficiency in Heterogeneous Hadoop Clusters by Adaptive Task Assignment," *in Proc. of ICDCS '15*, pp. 359–368, 2015.

[19] D. Cheng, X. Zhou, P. Lama, M. Ji, and C. Jiang, "Energy efficiency aware task assignment with dvfs in heterogeneous hadoop clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 70–82, 2018.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones." *in Proc. of NSDI'13*, pp. 185–198, 2013.

[21] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and Faster Jobs using Fewer Resources," *in Proc. of SoCC'14*, pp. 1–14, 2014.

[22] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," *in Proc. of OSDI'08*, pp. 29–42, 2008.

[23] S. Agarwal, S. Kandula, N. Bruno, M.-c. Wu, I. Stoica, and J. Zhou, "Reoptimizing Data-Parallel Computing," *in Proc. of NSDI '12*, pp. 281–294, 2012.

[24] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic Scheduling in Multi-Resource Clusters," *in Proc. of OSDI'16*, pp. 65–80, 2016.

[25] S. Jyothi, C. Curino, I. Menache, and et al., "Morpheus: Towards Automated SLOs for Enterprise Clusters," *in Proc. of OSDI'16*, pp. 117–132, 2016.

[26] S.Arimoto, S.Kawamura, and F.Miyazaki, "Bettering operation of dynamic systems by learning: a new control theory for servomechanism or mechatronic system," *in Proc. of CDC*, pp. 1064–1069, 1984.

[27] "Spark streaming," http://spark.apache.org/streaming.

[28] "Spark," http://spark.apache.org.

[29] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," *in Proc. of ICDEW'10*, pp. 41–51, 2010.

[30] S. Chintapalli, D. Dagit, B. Evans, and et al., "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming," *in Proc. of IPDPSW'16*, pp. 1789–1792, 2016.

[31] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," *in Proc. of EuroSys'10*, pp. 265–278, 2010.

[32] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using MapReduce," *in Proc. of SIGMOD'11*, pp. 985–996, 2011.

[33] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style Processing of Fast Data," *in Proc. of VLDB'12*, pp. 1814–1825, 2012.

[34] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The Design of the Borealis Stream Processing Engine." *in Proc. of CIDR'05*, pp. 277–289, 2005.

[35] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable Stream Computation in the Cloud," *in Proc. of EuroSys '13*, pp. 1–14, 2013.

[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm @ Twitter," *in Proc. of SIGMOD'14*, pp. 147–156, 2014.

[37] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," *in Proc. of SIGMOD'15*, pp. 239–250, 2015.

[38] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," *in Proc. of SoCC'10*, pp. 51–62, 2010.

[39] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," *in Proc. of OSDI'10*, pp. 1–15, 2010.

[40] P. Bhatotia, A. Wieder, R. Rodrigues, U. a. Acar, and R. Pasquin, "Incoop: MapReduce for incremental computations," *in Proc. of SoCC'11*, pp. 1–14, 2011.

[41] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic, "Adaptive scheduling of parallel jobs in spark streaming," *in Proc. of INFOCOM'17*, pp. 1–9, 2017.

[42] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing using Dynamic Batch Sizing," *in Proc. of SoCC'14*, pp. 1–13, 2014.

[43] H. Xu and W. Lau Cheong, "Optimization for speculative execution in big data processing clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 530–545, 2017.

[44] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming Stragglers in Approximation Analytics," *in Proc. of NSDI'14*, pp. 289–302, 2014.

[45] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," *in Proc. of EuroSys'11*, pp. 287–300, 2011.

[46] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," *in Proc. of SOSP'13*, pp. 69–84, 2013.

[47] D. Cheng, X. Zhou, P. Lama, J. Wu, and C. Jiang, "Cross-platform resource scheduling for spark and mapreduce on yarn," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1341–1353, 2017.