Lever: Towards Low-Latency Batch Stream Processing by Pre-Scheduling

***Abstract*—Many stream processing frameworks have been developed to meet the requirements of real-time processing. Among them, batch stream processing frameworks are widely advocated with the consideration of their fault-tolerance and high throughput. In batch 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 may inevitably result in high resource overhead or long job completion time. We notice that batch stream processing jobs are usually recurring with predictable characteristics. By exploring such feature, we present a pre-scheduling straggler mitigation framework called Lever. Lever first identifies potential stragglers and evaluates nodes' capability 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 the 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 batch stream processing system and outperforms traditional techniques significantly.**

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

I. Introduction

With the vast involvement of stream big data (e.g., stock market data, sensor data, social network data, etc.), quickly mining and analyzing stream big data is becoming more and more important. Many distributed stream processing frameworks such as Storm [1], Naiad [2], S4 [3], TimeStream [4], ELF [5] have been developed to meet the requirements of big data real time processing. Batch stream processing systems like HOP [6], Comet [7], HStreaming [8] and Spark Streaming [9] are widely adopted because these systems fully leverage the fault tolerance and high throughput properties of batch framework [10]. They propose to treat streaming workloads as a series of short batch jobs on small batches of streaming data, using batch processing engine such as MapReduce [11], Spark [12] to process micro-batch jobs.

Batch stream processing follows the well-known bulk synchronous parallel (BSP) model, where the job must wait until all the tasks complete. However, task execution time varies due to many negative factors, resulting in stragglers. In practice, straggler happens frequently and has been regarded as a major hurdle in achieving fast completion of latency-sensitive applications. In the production clusters at Facebook and Microsoft Bing, straggler tasks are on average 8 times slower than the median task in that job [13] [14]. Stragglers impacts the performance of small batch streaming jobs severely. On the one hand, small batch streaming jobs typically run all their tasks at less waves. Therefore, if one task is straggling, the whole job is significantly delayed. On the other hand, small batch streaming jobs are critically latency-sensitive. Straggling tasks will impact subsequent jobs' submission and execution. Therefore, mitigating or even eliminating stragglers remains a great challenge in batch stream processing.

Much effort has been devoted to migrating the stragglers. Existing representative straggler mitigation techniques are all post-scheduling and can be generally classified into two categories, i.e., reactive and proactive. Speculative execution [11] is a replication-based reactive straggler mitigation technique that spawns redundant copies of the stragglers, getting the results from the first completed task. It relies on a wait-speculate-re-execute mechanism, thus leading to delayed straggler detection and inefficient resource utilization. SkewTune [15] is another reactive approach without replication. It first need to spend much time on detecting data skew and then migrate some potential stragglers to other nodes. Although SkewTune avoids replicating tasks but is still a wait and speculate mechanism. Dolly [13] is a typical replication-based proactive strategy by full cloning of small jobs, avoiding waiting and speculation. But it incurs extra resources and it will become invalid when cluster resources become limited. Another proactive alternative without replication is Wrangler [14]. It learns to predict stragglers using a statistical learning technique based on cluster resource utilization counters and use these predictions to inform scheduling decisions. Wrangler is hysteric as it reacts only after some nodes show the sign of straggling. It can be seen that existing post-scheduling straggler mitigation approaches suffer from either high resource overhead for task replication or long task completion time due to delayed straggler detection.

Fortunately, batch streaming processing jobs which are run periodically as new data becomes available [9] are recurring with predictable characteristics, allowing us to carefully optimize the assignment of job input data before scheduling to avoid straggler. These job characteristics include input data distribution, straggler, task execution time, resource utilization and so on. Using these characteristics, we can plan ahead and pre-schedule how to allocate input data and task among nodes with different capability. It is expected that potential stragglers can progress as the same speed as others. Besides, better data locality can be also achieved by coordinating such data assignment with task placement on each node beforehand.

With this intuition, we present Lever, a pre-scheduling straggler mitigation framework that exploits the predictability

of recurring batch stream jobs to optimize the assignment of data. First, Lever observes and collects statistical information of previous jobs(e.g., task completion time on each node, the amount of data processed in previous jobs) when recurring jobs execute over time. Second, Lever identifies potential stragglers and evaluates node's computational capability by analyzing information from previous repetitions, then makes a capability-aware pre-scheduling plan about how to allocate data between each node. Finally, Lever uses this pre-scheduling plan as guidelines to repartition and dispatch input data in data receiving phase.

In summary, this paper makes the following contributions:

- We discuss the behaviors of existing straggler mitigation methods and identify the problems of existing straggler mitigation methods for batch stream processing systems.
- To better mitigate stragglers when processing small jobs, we present a pre-scheduling straggler mitigation framework named Lever which leverages the predictability of recurring jobs to pre-schedule input data.
- We implement a prototype of Lever based on Spark Streaming and verify its effectiveness through detailed evaluation. We also have contributed Lever as an extension of Apache Spark Streaming to the open source community. The experiments show that Lever can mitigate stragglers efficiently and improve the performance of stream applications significantly.

The rest of this paper is organized as follows. In Section II, we introduce the background and analyze the straggler problems in batch 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.

II. Background and Motivation

In this section, we first introduce the background of batch stream processing and stragglers. Then we deep into the problems existing in traditional straggler mitigation strategies when processing short stream jobs. We also briefly analyze the characteristics of recurring batch stream jobs. Finally, we outline the challenges in pre-scheduling straggler mitigation.

A. Background

Batch Stream Processing System treats a streaming computation as a series of deterministic batch computations on small time intervals [9]. As shown in Figure 1, Batch Stream Processing System (1) receives input data and divides continuous data stream into a series of small batches according to batch interval (a time granularity which is set according to application's response time), (2) when each batch interval arrives, batch jobs are generated for each batch, (3) these jobs would be repeatedly submitted to batch processing engine such as MapReduce or Spark and are re-run periodically to execute with distributed fault tolerance, data locality scheduling and load balancing provided by batch framework automatically. Latency sensitive, micro jobs/tiny tasks and recurring jobs are three dominant characteristics of these systems.

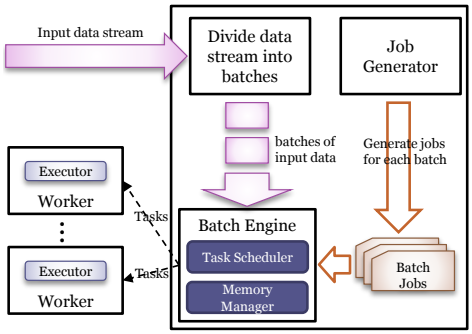


Fig. 1: Principles of Batch Stream Processing System

Stragglers have been a primary performance issue for batch stream processing system. Because the whole job must wait for the slowest task. These stragglers cause unproductive wait time for all the others, delaying job completion. Stragglers especially impact short batch stream jobs. On the one hand, short batch stream jobs are more latency-sensitive and require short response time. On the other hand, short batch stream jobs consist of just a few tasks. Such jobs typically get to run all their tasks at less waves. So, if one task straggles, the whole job is significantly delayed. Beyond these, if a job is so severely affected by stragglers that its completion time exceeds one batch interval, it will delay the next job's submission and execution. Stragglers can occur for many reasons, including hardware heterogeneity [16], data skew [15], hardware failures [17], energy efficient [18], resource contention and various OS effects.

B. Problem Analysis

Existing methods to straggler mitigation are post-scheduling methods. They fall short in multiple ways. Reactive approaches whether or not based on replication react after tasks have already slowed down. Proactive approaches which use replication incur extra resources. Proactive approaches without replication still take hysteretic actions after some nodes show stragglers' indications and need to migrate data at runtime. For short batch stream jobs, these approaches spend much reactive time to eliminate stragglers and cause other nodes to be idle over a period of time, wasting cluster computing cycles. In the following, we analyze four representative approaches. They are Speculative execution, SkewTune, Dolly and Wrangler on behalf of reactive approach with replication, reactive approach without replication, proactive approach with replication and proactive approach without replication respectively.

Reactive approaches rely on a wait-speculate-re-execute mechanism. Speculative execution [11] is a widely used reactive approach with replication for straggler mitigation. 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 killed. Taking Fig. 2(a) as an example, at t_b, T1 is marked as

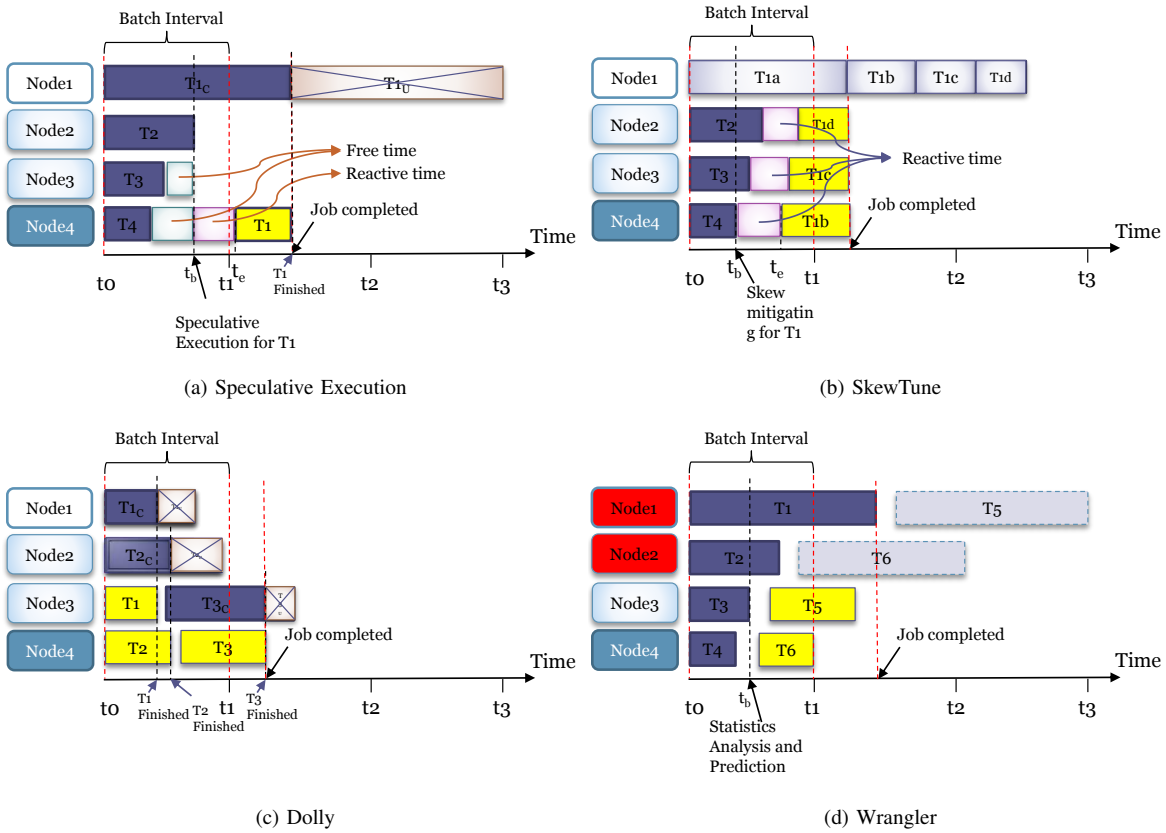


Fig. 2: Straggler mitigation under different strategies

stragglers. Then speculative execution launches a copy of T1 on node4. It migrates data block of Task T1 to node4 and starts executing at t_e. As we can see, this approach incurs long reactive time and free time for other nodes. It becomes inefficient for short batch stream jobs in two aspects. On one hand, a task must run for a significant amount of time before it can be identified as a straggler. By this time, other tasks of that job have made considerable progress already. On the other hand, the reactive procedure is time-consuming in terms of short stream job. A short stream job of several seconds response time is not allowed to spend several seconds on doing speculating and migrating. LATE [19] is an enhanced version of speculative execution using a notion of progress scores, but is still a wait-speculate-re-execute mechanism.

SkewTune [15] is still a reactive approach but without replication. As soon as 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 Fig. 2(b), when task T4 is completed at t_b, SkewTune's detection is triggered. Node1 is identified as straggler. T1 is selected to do skew mitigation due to its long remaining progress. Then SkewTune scans T1's remaining input data and repartitions T1's remaining work into T1b, T1c and T1d. Finally, T1b, T1c and T1d are scheduled and migrated to remote faster nodes node4, node3 and node2 respectively. From t_b to t_e, this procedure incurs approximately 2 seconds overhead for a task with 2MB's input data. This overhead grows linearly with the size of the task's input data. However, this overhead is intolerant for short stream jobs which must be completed in several seconds or sub-seconds. Furthermore, before SkewTune begins working, task T1 has already slowed down. Although SkewTune avoids replicating tasks but is still a wait-and-speculate mechanism.

Dolly [13] is a proactive strategy by full cloning of small jobs. Dolly launches multiple clones of every task of a job and only use the result of the clone that finishes first. As shown in Fig. 2(c), Dolly launches two clones of Task T1 and Task T2 in node1,3 and node2,4 respectively. After T1 and T2 complete, Dolly schedules T3 and also launches two clones of Task T3 in node3,4. Dolly gets the results from which finishes first such as Task T1 in node3 and kills other uncompleted clones such as Task T1 in node1. This not only amounts to wastage of resources, but also delays successor tasks.

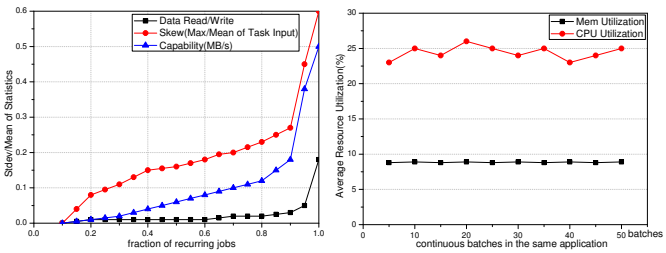
Unlike Dolly, Wrangler [14] is a proactive strategy without replication. It predicts stragglers using an interpretable linear modeling technique based on cluster resource usage counters and uses these predictions to inform scheduling decisions. For example, in Fig. 2(d), Wrangler analyzes statistics and make predictions about stragglers at t_b. Node1 and node2 are identified as stragglers. Then Wrangler schedules and migrates T5 in node1, T6 in node2 onto node3 and node4 respectively.

Although this approach can avoid successor possible stragglers, it remains taking hysteretic actions for short batch stream jobs. Because short batch stream jobs always consist of a few tasks and get to run at once. Furthermore, Wrangler can't act until some nodes show some indications of stragglers(eg. high cpu utilization, heavy memory access, network congestion). It also need to migrate data block at runtime regardless of data locality.

In summary, existing post-scheduling approaches fall short when processing short stream jobs. Reactive approaches act after stragglers have occurred. At that time, tasks have already slowed down. Replication-based proactive approaches incur extra resource and increase 2x load of the cluster. Proactive approaches without replication also take hysteretic actions after some nodes show some indications of stragglers. We need to design a pre-scheduling straggler mitigation strategy for short batch stream jobs which can eliminate stragglers as early as possible.

C. Recurring Batch Stream Jobs

Batch stream jobs are typically recurring with stable code and data properties. Many characteristics such as application logic, stragglers, resource utilization, scheduling and other operations present statistically similar to the previous execution when running on newer data of the same stream [20]. Figure 3(a) plots the average difference between statistics collected at the same location across different batches' jobs. We ran WordCount in our testbed and picked all the jobs from 50 batches, while the batch repeated in 2s' batch interval. The overall statistics are similar across jobs, the ratio stdev/mean is less than 0.2 for 70% of the locations. Figure 3(b) shows the resource utilization. These jobs cost almost the same amount of computational resources.



(a) Data Properties (b) Resource Utilization

Fig. 3: Stability of data properties and resource utilization across recurring jobs when running WordCount

Considering the following two points that 1) Straggler characteristics can often be accurately predicted in recurring jobs and 2) Job input data can be assigned in designated locations before scheduling to avoid straggler in advance, We propose Lever, a pre-scheduling straggler mitigation framework that continuously observes and learns the characteristics as recurring jobs execute over time and make a careful assignment about job input data ahead of execution. Doing so, Lever can efficiently reduce the number of queued tasks and avoid stragglers.

D. Challenges

We are faced with three challenges:

Challenge 1 How to identify potential stragglers?

Before pre-scheduling, Lever need to know which nodes will be stragglers in next batch. Unlike traditional ways which identify stragglers based on runtime-aware decisions, pre-scheduling takes actions before scheduling. Lever leverages the historical information of recurring jobs. But only analyzing historical information is not enough due to the variability of stream load. Accurately identifying potential stragglers is a prerequisite step for eliminating stragglers.

Challenge 2 How to define and determine node's capability?

Lever conducts pre-scheduling based on nodes' computational capability. To achieve this goal, it is necessary to define and determine node's capability. Traditional post-scheduling methods have resource utilization view at runtime and can schedule straggler tasks to that node which has free slots. However, in pre-scheduling solutions, it is impossible because we make pre-scheduling decisions before task scheduling. We should define a metric to determine how much data we should pre-schedule to that node. This metric represents to which degree this node will has free slots in future scheduling. So we introduce the notion of node's capability.

Challenges 3 How to select suitable helpers for data assignment?

To pre-schedule input data as evenly as possible to eliminate stragglers, Lever need to choose some helpers which are eligible to provide assistance. These helpers should be underloaded and have spare capability in next batch to deal with extra work assigned to them. As explained in Challenge 2, pre-scheduling don't have the perspective of runtime resource utilization. It brings great challenges when we make decisions about which nodes can afford extra load. How to choose suitable helpers with being unaware of task scheduling is a challenging problem.

To overcome these challenges, we design and implement a prototype of Lever.

III. System Design

In this section, we describe the design of Lever. Lever is designed to be API-compatible with the original system, providing the mitigation transparency and developer transparency. We begin with an overview of the system architecture, followed by introducing the system details.

A. System Overview

Figure 4 overviews the architecture of Lever. Lever periodically collects and analyzes the historical jobs' profiles of recurring batch streaming jobs. Based on these analysis, Lever pre-schedules job input data through three main steps, i.e. identify potential stragglers, evaluate nodes' capability and choose suitable helpers. Firstly, comparing each node's task finish time in previous batch, Lever can initially determine

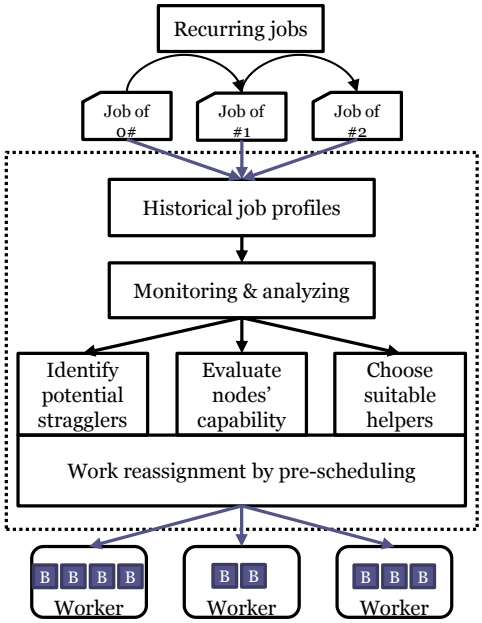


Fig. 4: Architecture of Lever

initial state of each node. Lever also estimates the tendency of input load. Combined with these two aspects, Lever conduct state transition to decide which nodes will behaviors as stragglers in next batch(for Challenge 1). Secondly, based on the fact that batch stream processing jobs are repetitive and periodic, Lever introduces Iterative Learning Control (ILC [21]), which is designed to do tracking control for systems that work in a repetitive mode to learn to estimate node's capability (for Challenge 2). Finally, considering that Lever don't have resource utilization view when pre-scheduling, Lever can choose all nodes or two least load nodes in faster group as helpers. Lever can switch between these two strategies adaptively to performance best(for Challenge 3).

We show the actions timing diagram of Lever in Figure 5. We differentiate pre-scheduling and post-scheduling according to their scheduling timing. Post-scheduling acts when processing but pre-scheduling operates on receiving. As shown in Figure 5, at last batch i.e. t_1 to t_2, Lever needs to collect and analyze previous jobs' execution information, identify potential stragglers and choose suitable helpers. At this batch i.e. t_2 to t_3, Lever pre-schedules input data according to prior pre-scheduling plan. When come to next batch i.e. t_3 to t_4, Lever schedules tasks according to data locality as usual. In the following, we present the details of Lever.

B. Identify Potential Stragglers

Lever predicts stragglers in the next batch according to recurring jobs' historical execution information combined with the load fluctuation of each node. The first step is to determine the initial stragglers. When last batch's jobs are completed, Lever collects and analyze the statistics of tasks' execution information in each node. The node i's finish time (NFT_i) is defined as the time from job's submission to the last task finish in this node. Then, Lever sorts node's list according to NFT_i in the descending order. It classify nodes into three categories according to two locations 0.25, 0.75 of node's list size. Nodes before 0.25×size are grouped into *straggler group*. Nodes after 0.75×size are grouped into *faster group*. The remaining are categorized as *median group*.

This classification can only represent the stragglers in the last batch and the second step is to add some transition conditions to predict which nodes' state will change in the next batch. First, Lever calculates the median finish time of *straggler group*, *median group* and *faster group* respectively, represented as *FTOS*, *FTOM*, *FTOF*. Second, Lever introduces degradation ratio *FTM* and *MTS*. *FTM* is defined as *FTOM*/*FTOF*. It means that if one node's NFT_i in *faster group* has increased by FTM, it will be moved from *faster group* to *median group* and vice versa. *MTS* is defined as *FTOS*/*FTOM*. It means that if one node's NFT_i in *median group* has increased by MTS, it will be moved from *median group* to *straggler group* and vice versa. We have shown straggler's state transition in Fig. 6.

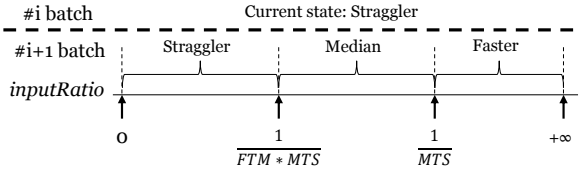


Fig. 6: An example for straggler's state transition

The third step is to do state transition, taking the load fluctuation into account. As stream load varies over time, last batch's stragglers may receive much less data and hence become faster in this batch. Also last batch's fasters may receive much more data to be overloaded and it will become straggler in this batch. Lever introduces *inputRatio* as a metric of stream load trend. It is defined as the quotient of new input rate in this batch and old input rate in last batch. We show the transition rules in Table I. Lever applies these rules to each node and decides final stragglers.

TABLE I: 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

C. Computational Capability Determination

In order to conduct capability-aware pre-scheduling, Lever evaluates each node's computational capability by leveraging

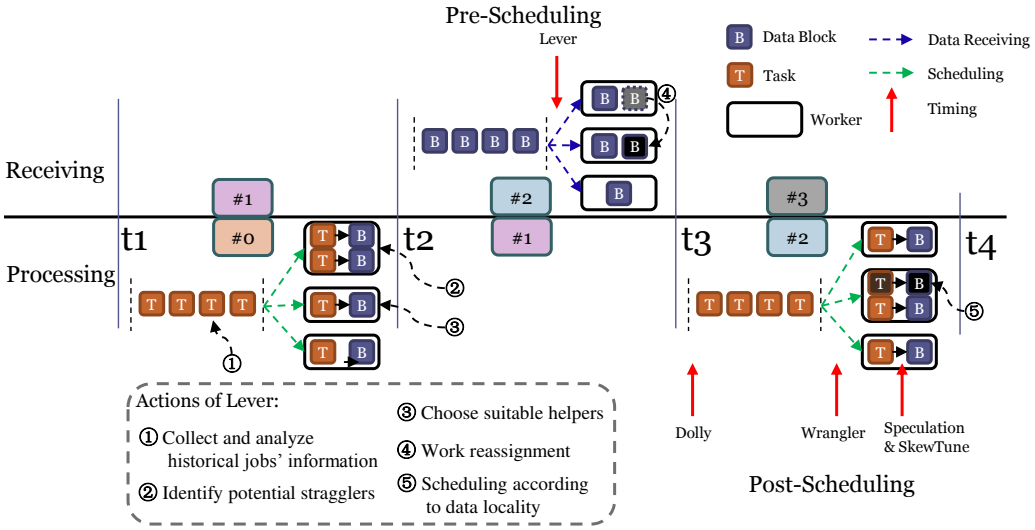


Fig. 5: Actions of Lever

the characteristics of recurring batch stream jobs.

1) Computational Capability: As shown in Section II-C, recurring batch stream jobs have stable data properties such as computational capability. By analyzing the previous execution, We can assess these metrics exactly in the subsequent execution.

Considering the periodicity of recurring batch stream jobs, we evaluate computational capability based on a well-known optimization technique, Iterative Learning Control (ILC [21]), which is designed to do tracking control for systems that work in a repetitive mode. This learning process uses information from previous repetitions to improve the estimation ultimately enabling a more and more accurate result can be found iteratively.

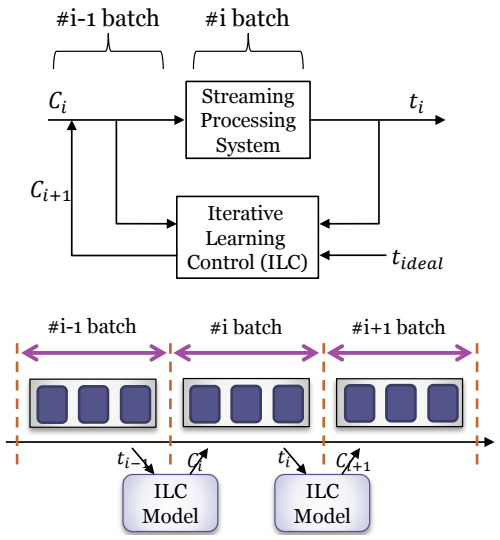


Fig. 7: The principle of evaluating computational capability with ILC.

Fig. 7 shows the principle of ILC algorithm. A simple control law for one 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 input capability to the stream processing system during the ith batch, Δt is node's finish time error during the ith batch and K is a design parameter representing operations on Δt. In Lever, K is set to C_i/t_i. t_{ideal} is the ideal node's finish time in the ith batch and can be got by computing the median node's finish time. At beginning, initial $C_{initial}$ is computed as the average of C_1, C_2 and C_3.

2) Theoretical Data Assignment: In the ideal case, every task should be completed simultaneously. The system should increase the amount of load in faster node and decrease those in straggler to minimize the makespan. Assume that there are n nodes, Let L_i and C_i denote the input load and the computational capability of node i respectively. Let L_i' denote the input load after been tweaked. Let t_i denote the finish time of node i. So we have:

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

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

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 $L_1\prime/C_1 = L_2\prime/C_2 = ... = L_n\prime/C_n$. The load L_i' after been tweaked can be denoted as:

$$L_i\prime = \frac{\sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} C_i} * C_i \tag{5}$$

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

$$\Delta L = L_i\prime - L_i = \frac{\sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} C_i} * C_i - L_i \tag{6}$$

D. Choose Suitable Helpers and Reassign Work

In Section III-B, we have grouped all workers into three groups. Nodes in faster group are eligible to provide assistance, and they can be helper which can afford a portion of stragglers' work. Nodes in straggler group are helpee to whom workers are eligible to provide assistance. Nodes in median group can do nothing. How to choose suitable helpers to provide assistance? We have two different strategies according to different situations and can adaptively choose one of two.

1) Strategies for Choosing Helpers: **All Strategy.** With this strategy, all nodes in faster group are selected as helpers. According to Section III-C, we can obtain the relatively ideal load distribution by the proportion of node's capability. Fig.8 shows the principle of all strategy. Assume that we have n helpers which ith helper is represented by the vector (C_i,L_i) and stragglers are the same. For each straggler with (C_s,L_s), its input data is assigned to those selected helpers according to each node's capability and load. The load share L_{sToj} which is dispatched to jth helper can be denoted as:

$$L_{sToj} = \frac{L_s + \sum_{i=1}^{n} L_i}{C_s + \sum_{i=1}^{n} C_i} * C_j - L_j \tag{7}$$

This strategy is suitable for the situation that there are few stragglers and fasters. Because we should take the overhead of partition and data transmission into account. For example, if we have n stragglers and m fasters, this strategy will produce $n * m$ data pieces and also $n * m$ socket collections. Huge amount of $n * m$ value will incur network congestion and impact current jobs' execution. So we add feedback control to Lever. When we detect that the response time of Lever is increasing after we adopt all strategy, then we will switch to two choice strategy.

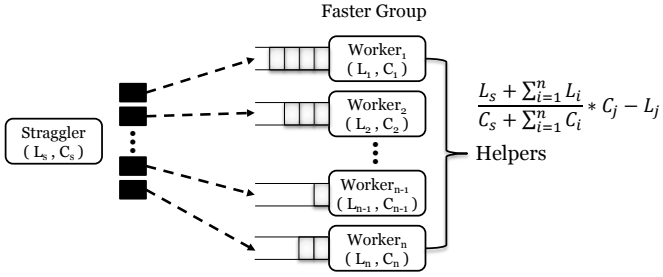


Fig. 8: All Strategy selects all nodes in faster group as helpers.

Two Choice Strategy. Different from all strategy, two choice strategy chooses two nodes in faster group as helpers. Lever intends to find out the nodes which have the large computational capability meanwhile have much less input data load as helpers. Fig. 9 shows the principle of two choice strategy. First, Lever sorts the nodes' list according to each node's C_i/L_i in the descending order. Then, Lever chooses the head two nodes as helpers and computes load share for each helper. Assume that the selected two nodes have vectors of (C_1,L_1) and (C_2,L_2), the straggler can be represented as (C_s,L_s). The load share L_{sToj} which is dispatched to jth helper can be denoted as:

$$L_{sToj} = \frac{L_s + L_1 + L_2}{C_s + C_1 + C_2} * C_j - L_j \tag{8}$$

After reassignment, Lever updates each node's C_i and L_i.

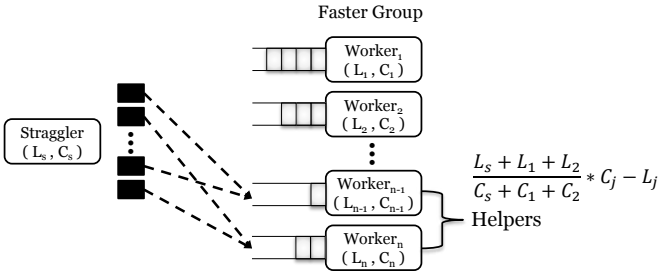


Fig. 9: Two Choice Strategy selects two nodes in faster group as helpers.

2) Adaptively Choose One of Two: The above two strategies can be used in different situations. Lever can adaptively choose one of two strategies according to cluster's current state. Obviously, when there are few stragglers, all strategy is much better than two choice strategy because we can pre-schedule input data as evenly as possible. However, if there are more and more stragglers and fasters, two choice strategy will behavior better because of avoiding heavy blocks partition and transmission. Lever combines the latency changes with the #stragglers×#fasters to make adaptive choice. If #stragglers×#fasters is larger than a observed value, Lever should select two choice strategy. Another case is that Lever uses all strategy last batch, however the applications' latency increases after we pre-scheduling, then Lever will switch to two choice strategy. The complete pseudo code of pre-scheduling algorithm is presented in Algorithm 1.

IV. System Implementation

In this section, we describe the implementation of Lever. We have implemented Lever based on a popular open-source distributed, batch stream processing system called Spark Streaming [22], which is an extension of cluster computing framework Apache Spark [23]. We choose Spark Streaming because it is a typical batch 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 by academic community and industrial community, and also been deployed on production clusters of hundreds of thousands of corporations. In this section, we describe the details of our system implementation. The source code of our system can be found at http://spark-packages.org/package/trueyao/spark-lever.

Figure 10 overviews the implementation of Lever. We add two components i.e. Runtime Analyzer and Scheduling Planner to master and three components i.e. Runtime Monitor, Partitioner and Dispatcher to worker, respectively. Runtime

Algorithm 1 Pre-Scheduling Algorithm

Procedure Identify Potential Stragglers
  Get the previous batches' job execution information
  Sort the nodes descending by their finish time
  Group the nodes into three groups(straggler, median, faster)
  Compute the input load's gradient
  Transform nodes' state according to transition rules
  Output final stragglers and fasters
End Procedure
Procedure Evaluate Computational Capability
  In ith batch, compute i+1th batch's capability
  $C_{i+1} = C_i + C_i/t_i * (t_{ideal} - t_i)$
  Assign corresponding load when receiving i+1th batch's data
  In i+1th batch, compute the next batch's capability
  $C_{i+2} = C_{i+1} + \frac{C_{i+1}}{t_{i+1}} * (t_{ideal} - t_{i+1})$
End Procedure
Procedure Choose Suitable Helpers and Reassign Work
if (#stragglers×#fasters<ρ) or (last batch use AllStrategy && latency decreases) **then**
  /* continue to adopt allstrategy */
  Choose all the nodes as helpers
  Compute the sum of helpers' capability and the sum of helpers' load
  $sumOfCapa = \sum_{i=1}^{n} C_i$ and $sumOfLoad = \sum_{i=1}^{n} L_i$
  for each straggler node **do**
    for each helper node **do**
      Compute the allocated load share
      $L_{stoi} = \frac{L_s + sumOfLoad}{C_s + sumOfCapa} \times C_i - L_i$
      Update each node's (C_i, L_i)
    end for
  end for
else
  /* adopt TwoChoiceStrategy */
  Sort the nodes' list according to each node's $\frac{C_i}{L_i}$ in the descending order
  Choose the head two nodes as helpers
  Compute the sum of helpers' capability and the sum of helpers' load
  $sumOfCapa = C_1 + C_2$ and $sumOfLoad = L_1 + L_2$
  for each straggler node **do**
    for each helper node **do**
      Compute the allocated load share
      $L_{stoi} = \frac{L_s + sumOfLoad}{C_s + sumOfCapa} \times C_i - L_i$
      Update each node's (C_i, L_i)
    end for
  end for
end if
End Procedure

Monitor which locates in each worker periodically detects worker node information such as load and processing speed, then reports to Runtime Analyzer. By analyzing these information combined with jobs' detailed execution information, Runtime Analyzer identifies the stragglers and evaluate each node's computational capability. These results will be passed to Scheduling Planner for pre-scheduling decisions. Scheduling Planner should make a reassignment plan about how to partition and pre-schedule stragglers' work to other nodes. Partitioner is responsible for partition straggler's excess work according to specified proportion which is derived from Scheduling Planner. Dispatcher distributes every piece of partitioned work in current straggler to corresponding helpers.

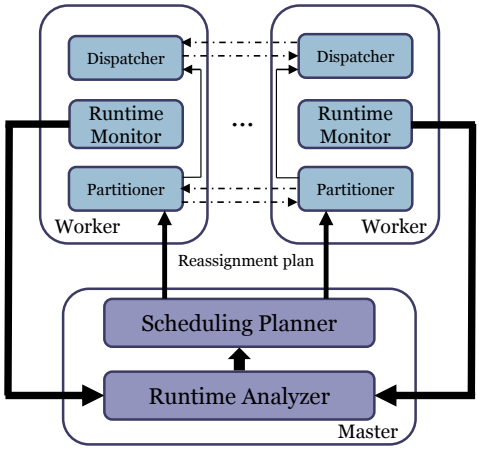


Fig. 10: Implementation of Lever

Runtime Monitor. *Runtime Monitor* locates in each worker and periodically detects worker node information. We add some functions in *executor*. When tasks are running in one executor, we can collect useful metrics such as task finish time and task's input data size through our function. We also add an accumulator in *worker*. If there is one task completed, these metrics about this task is encapsulated into a message which will be sent to accumulator after this. Accumulator gathers these messages and reports to *Runtime Analyzer* through an asynchronous RPC based on Akka.

Runtime Analyzer. Once one batch has finished, *Runtime Analyzer* begins to analyze the statistics information from *Runtime Monitor*. We create a new component which maintains a table i.e. *HashMap* in *master*. This table is used for recording and updating workers' information. This component mainly consists of many massage receiving operations and two essential functions. One is for updating table, another is for identifying stragglers and evaluating capability.

Scheduling Planner. This component receives messages from *Runtime Analyzer*. It includes a main function which is responsible for running our capability-aware pre-scheduling algorithm and a output table which records pre-scheduling data assignment plan. And we also add some callback functions in *JobScheduler* and *TaskScheduler* in order to get scheduling information feedback and adjust task scheduling.

Partitioner. We modify the *BlockGenerator* and the *ReceiverSupervisorImpl* to implement the Partitioner. We add a partition function *splitBlockBuffer* to divide the receiving buffer into specified data share. And these fragments are delivered to *BlockManager* as so called *block* first(register the meta data such as block size and location to *BlockManager*).

Dispatcher. Function *reallocateBlock* is the core of Dispatcher. This component carry out two things. One is getting the host name which data piece will be assigned to from the assignment plan. And another is invoking the *blockTransferService* to transfer block. We modify the system call *uploadBlockSync* in Lever to implement the remote block assignment.

V. Experimental Evaluation

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

A. 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 eight dual-core Intel Xeon E5-2670 2.6 GHZ CPUs(a total of 16 cores), 64GB memory, and 210GB disks. The second kind of machines consists of two dual-core 1.2 GHZ Intel CPU, 16GB 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.

Workloads. Table II describes the benchmarks used in our experiments. Similar to previous work [9], we choose three typical applications i.e. Grep, WordCount, and TopK. In addition, we also use the HiBench [24] benchmark including Identity, Sample and Projection. We rewrite the HiBench [24] benchmark based on receiver model.

Other configurations. In all our experiments, we use default data locality wait time(3s), therefore Delay Scheduling [25] doesn't not interfere with our experimental results. For speculation, we set the speculation interval, speculation quantile(default 0.75) and speculation multiplier(default 1.5) as 100ms, 0.5, 1 respectively. These settings can let speculative execution perform much best in our environment. We implement Skewtune, Dolly and Wrangler's prototypes. We adopt local scan for skewtune. We also set wait duration of delay assignment ω as 200ms in Dolly. For Wrangler, we set confidence measure ρ as 0.7 and set interval(decides how frequently a snapshot of node's resource usage counters is collected) Δ as one batch.

Unless specified otherwise, we use 2s' batch interval and 100-bytes input records. Stragglers are on average 8 times slower than the median task in each job. All input data blocks only have one replica. We don't 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.

B. Job Completion Time

We first test the improvement in completion time using Lever. Figure 11 reports the normalized job completion time when compared with other strategies. Lever can significantly improve performance when running Projection, Grep, WordCount and Topk. But Lever improves a little for Identity and Sample relatively. It depends on the type of workloads. Identity and Sample are of simple execution logic. Straggler has little impact on these workloads.

Compared to baseline, Lever improves the average job completion time by 32.31%, 30.72%, 37.54% and 42.19% for Projection, Grep, WordCount and Topk, respectively. Speculation and Skewtune works much worse in our experiments because they need to speed a long time on detecting stragglers and data skew. Lever pre-schedules job input data before task scheduling and avoid 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) and Topk(by up to 34.48%, 41.24%, 34.48% and 31.33% compared to Speculation, Skewtune, Dolly and Wrangler).

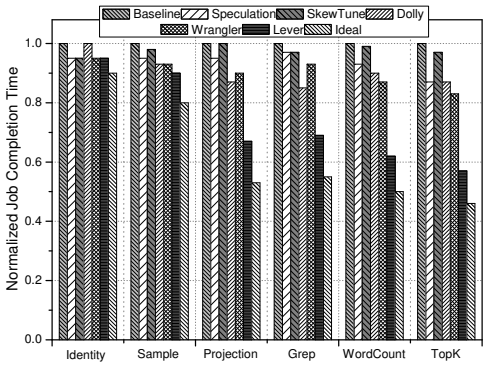


Fig. 11: Normalized job completion time when running benchmarks under different techniques

Figure 12 shows a detailed analysis of task completion time in three types of nodes. We average task's completion time of faster group, median group and straggler group. Task's completion time in each node decides the makespan of this job. As we can see, straggler is quite remarkable under traditional methods. It means that these techniques cannot eliminate stragglers very well. As a result, the straggler nodes are load-heavy all the batch and the fasters are idle for a long time. It is due to their fumbling reaction to stragglers. In the ideal case, all the nodes should complete their tasks at almost the same time. Lever behaviors much better because it balances data skew when receiving data. So when tasks are running, all nodes can progress at almost the same rate.

Table III lists the node's idle time when a batch is completed. Baseline and Skewtune waste more than 3s for faster and median nodes totally in a 2s' batch interval. Speculation and Wrangler are idle for about 2s. Much of this time is spent on detection and migration. This is the root cause of why they perform much worse than others. Although Dolly wastes less time, it clones tasks, which means that it need to complete 2x amount of work. Lever also has idle time as long as 0.625s. This is because Lever only takes the straggler and faster into account regardless of median group's node. For median, it is difficult to estimate whether it will be faster or slower than stragglers after we redistribute load.

In summary, Lever saves a lot of idle time through pre-

TABLE II: Benchmarks

Benchmark	Description	Complexity	Resource Preference	Source
Identity	Simply reads input and takes no operations on it	Single Step	None	HiBench [24]
Sample	Samples the input stream according to specified probability	Single Step	None	HiBench [24]
Projection	Extracts a certain field of the input	Single Step	Network Intensive	HiBench [24]
Grep	Checks if the input contains certain strings	Single Step	Network Intensive	DStream [9]
WordCount	counts the number of each word in input text	Multi Step	CPU Intensive	DStream [9]
Topk	Finds the k most frequent words over the specified window	Multi Step	CPU Intensive	DStream [9]

TABLE III: Node's idle time for faster and median

Techniques	Baseline	Speculation	Skewtune	Dolly	Wrangler	Lever	Ideal
Idle time	3.243s	2.363s	3.137s	0.875s	2.224s	0.625s	0.055s

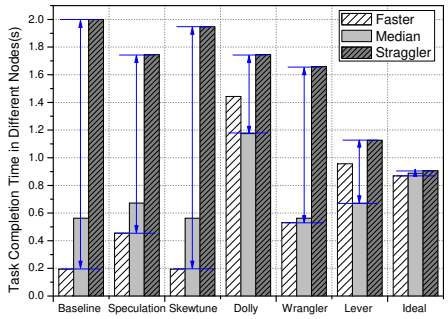


Fig. 12: Task completion time in three types of nodes when running WordCount

scheduling data assignment, thus avoiding detection, migration and cloning. So Lever can mitigate stragglers and improve the performance significantly.

C. 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 greatly than batch interval. We also test other techniques' effectiveness under this situation.

From the test result in Figure 13, we observe that both Lever and Wrangler can achieve much better performance. Baseline, Speculation, Skewtune and Dolly can't process subsequent jobs timely because their cloning and hysteresis reaction aren't not enough, leading a large number of subsequent jobs are queueing in scheduler's waiting queue. The scheduling delay will increases monotonously.

Although Wrangler can go on processing subsequent jobs, Wrangler 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 nodes being overloaded.

Lever avoids this problem by analyzing recurring jobs' load fluctuation in previous batches. As shown in Figure 13, Lever can converge to an ideal latency in six batches. The estimation for capability by using ILC also returns to normal in six batches after experiencing a short period of fluctuations.

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

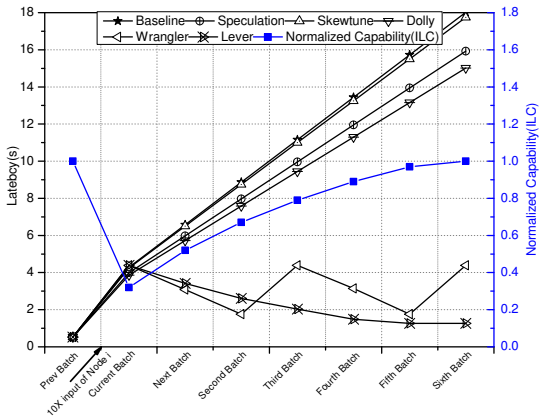


Fig. 13: Latency and Normalized Capability when running WordCount under burst load

D. Data Locality and Queued Tasks

In this test, we conduct a statistical analysis about data locality and queued tasks which 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 14, 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

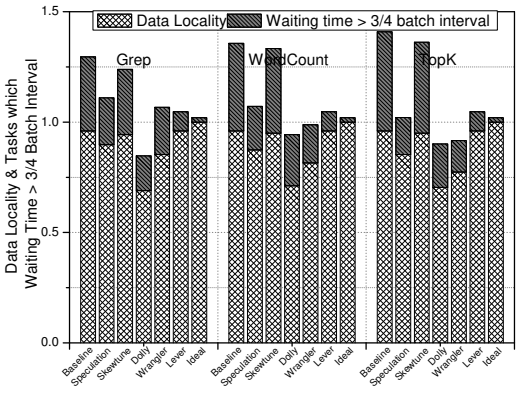


Fig. 14: Data Locality and Tasks which waiting time is greater than 3/4 batch interval when running Grep, WordCount and Topk

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't react to straggler and data skew quickly.

In the ideal case, all nodes execute local tasks and there aren't tasks queued for 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 capability and Lever 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 which waiting for a long time.

E. All Strategy or Two Choice Strategy

In this test scenario, we compare two helper-choosing strategies when faced with different straggler situations. We varies the number of stragglers and the number of fasters to evaluate the advantages and disadvantages of two methods.

The experimental results presented in Figure 15 show that when there are few stragglers in current cluster, allStrategy is much better than twoChoiceStrategy. However, if there are more and more nodes will be stragglers, twoChoiceStrategy behave much better. We give an example to elaborate the reason. Assume that we have n stragglers and m fasters, allStrategy produces n×m partitions to migrated and rebalanced, but twoChoiceStrategy only produces n×2 fragmentations. Although allStrategy makes load well-distributed, which accompany with it is the increasing overhead in partition and network transformation.

In our experiments, when there are 8 stragglers and 10 fasters, twoChoiceStrategy begins to perform better than allStrategy. It means that when the observed value ρ is larger than 80, then Lever should choose twoChoiceStrategy. As shown in Figure 15, Lever can select helper-choosing strategies adaptively according to straggler situations and latency fluctuation.

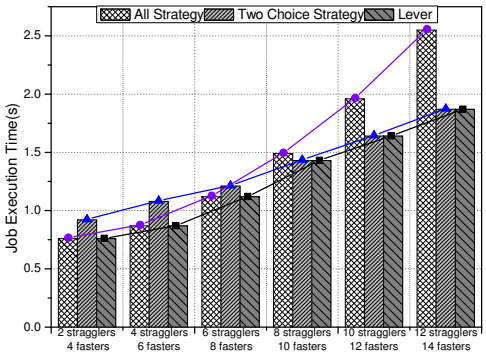


Fig. 15: Results under different straggler situations using two strategies

In summary, allStrategy is suitable for few stragglers and few fasters and twoChoiceStrategy is more suitable for many stragglers or many fasters. Lever can choose one of the two adaptively to ensure the performance.

VI. Related Work

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

Batch Stream Processing and Incremental Data Processing Systems. Batch stream processing systems collect received data into batches and periodically process them using MapReduce-style batch computations. The typical systems include Comet [7] which is structured on DryadLINQ, HOP [6] which leverages the power of batch framework MapReduce, and Spark Streaming [9] 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 [26] and [27] intent to modify batch framework to adapt to the requirements of stream processing. Other stream processing systems such as Borealis [28], TimeStream [4], Naiad [2], Storm [29] and Heron [30] 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 [31],Percolator [32] and Incoop [33] allows 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.

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 scheduling methods are Delay Scheduling [25], LATE [19], and Tarazu [34]. Delay Scheduling tries to maintain data locality by later decision. LATE speculates slow tasks using a notion of progress scores. Tarazu designs communication-aware load balancing of map computation, communication-

aware scheduling of map computation, and predictive load balancing of reduce computation to respectively prevent shuffle-critical tasks stealing, interleave remote tasks with local ones, and skew the intermediate key distribution among the reduce tasks. They are reactive ways for batch processing. The typical straggler mitigation approaches are Speculative Execution [11], Mantri [17], Dolly [13], GRASS [35] and Wrangler [14]. Speculative Execution marks slow tasks as stragglers and launch 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, it proposes full cloning of small jobs by delay assignment. Dolly don't 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, 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 [36], SkewTune [15]. Scarlett replicates block 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.

Load Balancing for Stream Processing. Many load balancing techniques for stream processing [37], [38], [39], [40] have been proposed. Flux [37] monitors the load of each operator, ranks servers by load, and migrates operators from the most loaded to the least loaded server, from the second most loaded to the second least loaded, and so on. Aurora and Medusa propose policies to migrating operators in stream processing [38]. Borealis uses a similar approach but it also aims at reducing the correlation of load spikes among operators placed on the same server [39]. [40] introduces a new stream partitioning scheme that adapts the classical power of two choices to a distributed streaming setting by leveraging two techniques: key splitting and local load estimation. These techniques are designed for continuous operator stream processing, not suitable for batch stream processing.

VII. Conclusion

This paper presents Lever, a pre-scheduling straggler mitigation framework that exploits the predictability of recurring batch 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' capability. Furthermore, Lever carefully chooses helpers and optimizes the assignment of the job input data according to each node's capability proportion. Lever has been implemented on the top of Spark Streaming. Lever is also open source and is now included in Spark Packages at http://spark-packages.org/package/trueyao/spark-lever. We conduct various experiments to validate the effectiveness of Lever. The experimental results demonstrate that Lever reduces job completion time by 30.72% to 42.19% and outperforms traditional techniques significantly.

In future work, we plan to enhance the accuracy for estimation of node's computational capability. A possible approach is to introduce a new machine learning method to evaluate each node by collecting the node-level features of hardware configuration and resource utilization such as CPU, memory, disk and network. By using a statistical model to analyze these information, we can get a relatively more accurate result to direct capability-aware pre-scheduling. Another direction in the future is that we will consider the influence of shuffle-heavy tasks and how to balance load at shuffle stage. It will extend the usage of Lever.

Acknowledgment

This work was supported by National Science Foundation of China under grant No.61472151 and No.61232008, National 863 Hi-Tech Research and Development Program under grant No.2014AA01A302 and No.2015AA01A203, the Fundamental Research Funds for the Central Universities under grant No.2015TS067.

References

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

[2] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 439–455, 2013.

[3] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," *IEEE International Conference on Data Mining Workshops (ICDMW'10)*, pp. 170–177, 2010.

[4] 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," *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 1–14, 2013.

[5] L. Hu, K. Schwan, H. Amur, and X. Chen, "ELF: efficient lightweight fast stream processing at scale," *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (ATC'14)*, pp. 25–36, 2014.

[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*, pp. 21–35, 2010.

[7] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pp. 63–74, 2010.

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

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pp. 423–438, 2013.

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

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pp. 137–149, 2004.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing (HotCloud'10)*, pp. 10–16, 2010.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones." *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pp. 185–198, 2013.

[14] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and Faster Jobs using Fewer Resources," *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*, pp. 1–14, 2014.

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating Skew in Mapreduce Applications," *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*, pp. 25–36, 2012.

[16] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. a. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," *Proceedings of the 3th ACM Symposium on Cloud Computing (SoCC '12)*, pp. 1–13, 2012.

[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," *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (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," *IEEE International Conference on Distributed Computing Systems (ICDCS '15)*, 2015.

[19] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments." *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, pp. 29–42, 2008.

[20] S. Agarwal, S. Kandula, N. Bruno, M.-c. Wu, I. Stoica, and J. Zhou, "Re-optimizing Data-Parallel Computing," *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pp. 281–294, 2012.

[21] S. S.Arimoto and F.Miyazaki, "Bettering operation of dynamic systems by learning: a new control theory for servomechanism or mechatronic system," *Proceedings of 23rd Conference on Decision and Control*, pp. 1064–1069, 1984.

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

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

[24] "Hibench," http://github.com/intel-hadoop/HiBench.

[25] 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," *Proceedings of the 5th European conference on Computer Systems (EuroSys'10)*, pp. 265–278, 2010.

[26] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using MapReduce," *Proceedings of the 2011 international conference on Management of data (SIGMOD'11)*, pp. 985–996, 2011.

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

[28] 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." *Conference on Innovative Data Systems Research (CIDR'05)*, pp. 277–289, 2005.

[29] 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," *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD'14)*, pp. 147–156, 2014.

[30] 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," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pp. 239–250, 2015.

[31] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pp. 51–62, 2010.

[32] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pp. 1–15, 2010.

[33] P. Bhatotia, A. Wieder, R. Rodrigues, U. a. Acar, and R. Pasquin, "Incoop: MapReduce for incremental computations," *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*, pp. 1–14, 2011.

[34] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu : Optimizing MapReduce On Heterogeneous Clusters," *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, pp. 61–74, 2012.

[35] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming Stragglers in Approximation Analytics," *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pp. 289–302, 2014.

[36] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," *Proceedings of the 6th European conference on Computer Systems (EuroSys'11)*, pp. 287–300, 2011.

[37] M. a. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," *International Conference on Data Engineering (ICDE'03)*, pp. 25–36, 2003.

[38] M. Cherniack, H. Balakrishnan, and M. Balazinska, "Scalable Distributed Stream Processing," *Conference on Innovative Data Systems Research (CIDR'03)*, pp. 811–825, 2003.

[39] Y. Xing, S. Zdonik, and J. H. Hwang, "Dynamic load distribution in the borealis stream processor," *International Conference on Data Engineering (ICDE'05)*, pp. 791–802, 2005.

[40] M. Anis, U. Nasir, G. De, F. Morales, N. Kourtellis, and M. Serafini, "The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines," *International Conference on Data Engineering (ICDE'15)*, pp. 137–148, 2015.