Stateful Load Balancing for Parallel Stream Processing

Qingsong Guo[1] and Yongluan Zhou[2]

[1]North University of China, Taiyuan, China
qingsongg@gmail.com
[2]University of Copenhagen, Copenhagen, Denmark
zhou@di.ku.dk

Abstract. Timely processing of streams in parallel requires dynamic load balancing to diminish skewness of data. In this paper we study this problem for stateful operators with key grouping for which the process of load balancing involves a lot of state movements. Consequently, load balancing is a bi-objective optimization problem, namely MINIMUM-COST-LOAD-BALANCE (MCLB). We address MCLB with two approximate algorithms by a certain relaxation of the objectives: (1) a greedy algorithm ELB performs load balancing eagerly but relaxes the objective of load imbalance to a range; and (2) a periodic algorithm CLB aims at reducing load imbalance via a greedy procedure of minimizing the covariance of substreams but ignores the objective of state movement by amortizing the overhead of it over a relative long period. We evaluate our approaches with both synthetic and real data. The results show that they can adapt effectively to load variations and improve latency efficiently comparing to the existing solutions whom ignored the overhead of state movement in stateful load balancing.

1 Introduction

Timely processing of big streaming data on a cluster of commodity machines is the major concern for a *stream processing engines* (SPE) like Storm [1]. Usually, streaming computations are organized as an operator graph in which vertices stand for operators and an arc in the graph represents the data streams flowing between a pair of operators called *producer* and *consumer* respectively. To handle the deluge of data, a SPE exploits data parallelism that splits a stream into a number of disjoint substreams so that them can be processed independently by a collection of parallel instances.

Process a stream in parallel relies on the grouping scheme for dispatching tuples to the instances of its consumer. Typically, there are two primitives of our interest: (1) *shuffle grouping* and (2) *key grouping* [10]. In shuffle grouping, tuples are randomly routed to downstream instances. It fits for stateless operators like *map* and *filter*, which are content-oblivious so that a tuple can be processed by any instances. In contrast, the key grouping partitions a stream into a number of substreams based on the key, i.e., a set of attributes, where tuples have equal values on key will be dispatched to the same instances. Stateful operators like **window-join** are content-sensitive since tuples with the same value should be processed by the same instance. Therefore, key grouping is preferable for stateful operators.

In this paper, we concern the problem of balancing load for stateful operators implementing key grouping. For a stateless operator with shuffle grouping, its load can be balanced evenly in a round-robin manner. However, it becomes much challenging in our context since the key grouping results in load imbalance. A substantial feature of stream processing is that data is in a state of ceaseless change [13,16,15]. Load variations like fluctuation of data rate and change in data distribution are ubiquitous, especially for such applications with their sources geographically located. If the load distribution is skewed on the partition key, the number of tuples handled by instances vary greatly. The computation will often be situated in an erratic state if we do not react to the imbalance, which is a disaster for processing latency if the state lasts for a long time.

Load balancing has received much attention in distributed stream processing [2,15,16]. Xing et al. [16] presented a correlation-based load distribution policy for a homogeneous shared nothing cluster. They focused on balancing load for a whole operator graph with an implicit assumption that every operator is not parallelized. In contrast, we focus on balancing load for a single operator with very high volume of load. In addition, load balancing has been also addressed for stateless operators with key grouping [10]. The impact of processing state has been widely studied in parallel stream processing [12,14], but it rarely brings about any attention to load balancing. In the presence of state, it involves a lot of state movements in load balancing because we have to change the allocations for many substreams. This problem is referred to as *stateful load balancing* and we formally define it as MINIMUM-COST-LOAD-BALANCE (MCLB). It associates two objectives: (1) minimize imbalance of all instances as much as possible; and (2) minimize the state movements as many as possible.

Unfortunately, the two objectives of MCLB can not be optimized consistently since they conflict with each other. Timely processing of data stream relies on efficient algorithms to address this dilemma. We propose two approximate algorithms for MCLB by relaxing the constraint on load imbalance: (1) ELB that balances the load eagerly, and thus has expensive cost of state movements; and (2) an algorithm CLB based on a procedure of minimizing correlations, that performs the load balancing periodically, where the cost for state movement is amortized and which is negligible when the period length is long enough. We evaluate the algorithms, with both sythetic streams and real datasets, and compare them with the exiting solutions. The experimental results justify the advantage of our solutions.

1.1 Related Work

Load balancing has received much attention in the last decade for its application in the peer-to-peer system [4] and cloud computing [11]. These approaches are static and hence are insufficient for a streaming scenario in which data is in ceaseless change [5]. Madsen et al. [8,9] recognized the problem of stateful load balancing while optimizing cluster utilization and minimizing latency for parallel stream processing. They modeled it as a Mixed-Integer Linear Program(MILP) problem and derive a solution with a MILP solver by incorporating the overhead of state movements into the constraints. In addition, there are three existing work that are analogous to our work [13,3,10].

Shah et al. [13] studied how to process a single continuous query operator on multiple shared-nothing machines. In this work, load imbalance is distinguished into *short-term imbalance* and *long-term imbalance*. Load balancing is in charge by an operator Flux that encapsulates adaptive partitioning and routing. To reduce the state movements, Flux sorts the sites in descending order of load and pairs them together, where load balance is realized by moving partitions around the sites in each pair. However, the parallelism in Flux is fixed and the cost for state movements has also not been quantified.

Nasir et al. [10] investigated the load balancing problem for stateless operators by applying the "*power of two choices*" approach. Their solution, namely Partial Key Grouping (PKG), improves the performance by mapping each key to two distinct substreams and forwarding each tuple to the less loaded of the two substreams. This approach can not be applied directly to stateful operator, because we need an extra operator to consolidate the partial results.

Gedik [3] proposed a partition scheme that is close to our solution. Stream is split with a partition function $\langle \mathcal{H}_t, \mathcal{H}_c \rangle$, which is a hybrid of consistent hash and explicit mapping, for multidimensional load balancing in stateful parallelization. This strategy can be applied for dynamic load balancing, but it has two drawbacks: (1) it has to reconstruct a new partition function after each process, which introduces new overhead for processing latency; and (2) it will result in expensive state migration since it uses a hash function to rebalance the load as we addressed.

2 Stateful Load Balancing

2.1 Problem Statement

A streaming computation is usually organized as an operator graph [1].Each operator implements a bunch of predefined processing logic, such as *join*, *aggregate*, *filter*, or *user-defined functions*. A stream s can be written as an operator pair (u_s, o_s), where u_s and o_s are the producer and consumer of it respectively. At runtime the consumer o is parallelized into a number of instances $\mathcal{I} = \{o^1, \ldots, o^n\}$, where $n \in \mathbb{N}^+$ is the parallelism. Stream s associates with a key k, the domain of the partition key k_u is split into p partitions with a hash function $\mathcal{H}(K_u) : \mathcal{D} \rightarrow [1 \ldots p]$, which separates s into non-overlapping substreams $\mathcal{S} = \{s^1, \ldots, s^p\}$, where $p \gg n$ and $p = \mathcal{O}(n)$. If o is stateful, then its processing state PS is also split into p partitions $ps = \{ps_1, ps_2, \ldots, ps_p\}$. A parallel processing of s is defined by the assignment $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{I}$.

Stateful load balancing. For a stateful operator with key grouping, the number of tuples processed by each instance vary greatly if the distribution on the key is skewed. It is inevitable to balance load for instances. We focus on load balancing for a single operator o rather than the whole query graph. For convenience of discussion, we suppose that operator o has a unique input stream s. The assignment $\mathcal{F}$ changes at runtime so as to handle load variations. A state partition ps_i should be moved to another instance if the allocation of substream s^i has been changed. Therefore, the process of load balancing involves a lot of state movements and we call it as *stateful load balancing*.

2.2 Minimum Cost Load Balancing

Decision on LB relies on statistics about data rate, load distribution, and state distribution. Statistics are collected periodically over *statistic windows* of length Δ. We use a histogram $Y_t = (y_{1t}, y_{2t}, \ldots, y_{pt})^T$ to record the load distribution of $s_1 \ldots s_p$ in the

t-th window, where y_{it}, $i = 1 \ldots p$, is the number of tuples of s_i arrived in this window. Other statistics about s like the mean $\bar{y}_t$ and the variance $var(Y_t)$ of Y_t can be derived accordingly. Given a histogram Y_t and an assignment $\mathcal{F}_1$, we can measure the load imbalance and the number of state movements for the t-th statistic window.

Load imbalance. Encoding the assignment $\mathcal{F}_1$ as a matrix $\boldsymbol{A} = [a_{ij}]_{p \times n}$, where a_{ij} is a binary variable such that $a_{ij} = 1$ if substream s_i is assigned to instance o^j and $a_{ij} = 0$ otherwise. Since each substream only can be processed by an instance, we have $\sum_{j=1}^{n} a_{ij} = 1$. Let $L_t = (l_{1t}, l_{2t}, \ldots, l_{nt})^T$ be the *load vector* for instances $(o^1, \ldots, o^n)$ in the t-th window, then it is given by a linear transformation $L_t = \boldsymbol{A}^T Y_t$. If $\mathcal{F}_1$ is a balanced assignment, then $\boldsymbol{A}^T Y_t = \boldsymbol{\bar{l}_t}$, where $\boldsymbol{\bar{l}_t} = (\bar{l}_t, \bar{l}_t, \ldots, \bar{l}_t)^T$ and $\bar{l}_t = \frac{1}{n} \sum_{i=1}^{p} y_{it}$ is the *average load* in the t-th window.

Much work [13] defines the load imbalance in the t-th window as the difference between the maximum and the average load of instances, i.e., $\max_i(l_{it}) - \bar{l}_t$. But this value is insufficient to reflect the load distribution, which plays an essential role in changing the assignment. Alternatively, we use the *variance* of load vector $L_t = (l_{1t}, l_{2t}, \ldots, l_{nt})^T$ to measure the load imbalance in the t-th window. That is,

$$var(L_t) = \frac{1}{n} \sum_{i=1}^{n} (l_{it} - \bar{l}_t)^2, \tag{1}$$

where $L_t = \boldsymbol{A}^T Y_t$, and $\bar{l}_t$ is the mean of L_t, i.e., $\bar{l}_t = E(L_t) = \frac{1}{n} \sum_{i=1}^{n} l_{it}$.

State movement. Consider an adaptation and $\mathcal{F}_2$ is a new assignment. A state partition ps_i, $i = 1 \ldots p$, will be moved to another instance if the allocations given by two assignments are different, i.e., $\mathcal{F}_1(s_i) \neq \mathcal{F}_2(s_i)$. Let $\boldsymbol{x} = (x_1, \ldots, x_p)^T$ be a vector of binary variables, where $x_i = 1$ if $\mathcal{F}_1(s_i) \neq \mathcal{F}_2(s_i)$ and $x_i = 0$ otherwise. Let $\boldsymbol{d} = (d_1, \ldots, d_p)^T$ be the *state distribution* at present, where d_i is the number of tuples in ps_i. Then the number of state movements $\psi(\mathcal{F}_1, \mathcal{F}_2)$ in this load balancing is:

$$\psi(\mathcal{F}_1, \mathcal{F}_2) = \boldsymbol{x} \cdot \boldsymbol{d} = \sum_{i=1}^{p} x_i d_i \tag{2}$$

Given a set of substreams $\mathcal{S} = \{s_1, \ldots, s_p\}$ and a number of instances $\mathcal{I} = \{o^1, \ldots, o^n\}$, we consider a load balancing that replaces the current assignment $\mathcal{F}_1 : \mathcal{S} \to \mathcal{I}$ with a new one $\mathcal{F}_2$. The decision of load balancing must rely on statistics of historical data. Assuming we have a sequence of histograms $\boldsymbol{Y} = (Y_1, \ldots, Y_m)$, $m \in \mathbb{N}$, over the latest m statistic windows. We have a sequence of load vectors $\boldsymbol{L} = (L_1, \ldots, L_m)$, where the load vector L_j is given by $L_j = \boldsymbol{A}^T Y_j$. The overall imbalance over the statistic windows is $\hbar(\mathcal{F}_1) = \sum_{j=1}^{m} var(L_j)$. In addition, the cost of state movements of replacing $\mathcal{F}_1$ with $\mathcal{F}_2$ is given by Eq. 2, which quantifies the amount of communication required for approaching the load balancing. Therefore, the stateful load balancing is to compute an assignment that minimize both simultaneously. We denote this problem as MINIMUM-COST-LOAD-BALANCE (MCLB).

MCLB is a *bi-objective optimization* problem and it has been proved to be NP-hard. It is apparent that the two objectives conflict with each other: (1) to minimize $\psi(\mathcal{F}_1, \mathcal{F}_2)$, one hopes to change the assignment as less as possible; (2) to minimize $\hbar(\mathcal{F}_2)$ one needs more movements for which one can try more possible plans so as to balance the load. Therefore we cannot compute a feasible solution that minimizes both objectives simultaneously. Instead, we present two approximate algorithms for MCLB.

Algorithm 1: Eager Load Balancing (ELB)

```
Input: The rurrent assignment $\mathcal{F}$, Histogram $Y_t = (y_{1t}, \ldots, y_{pt})^T$
Output: New assignment
Initialization: $OI \leftarrow \emptyset, UI \leftarrow \emptyset, R \leftarrow \emptyset, PQ \leftarrow \emptyset$ ;
/* Phase 1: preparing                                                  */
$(l_{1t} \ldots l_{nt})^T \leftarrow \boldsymbol{A} Y_t, w \leftarrow \sum_{j=1}^{n} l_{jt}$;
$\pi \leftarrow \lceil \frac{2w}{u+v} \rceil, \bar{l}_t \leftarrow \frac{w}{\pi}$ ;
$o^1 \ldots o^n \leftarrow$ sort $\mathcal{I}$ in descending order of loads ;
if $\pi > n$ then
    $o^{n+1} \ldots o^{\pi} \leftarrow$ initialize $\pi - n$ instances with load of 0;
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{o^{n+1}, \ldots, o^{\pi}\}$
if $\pi < n$ then
    $R \leftarrow o^{\pi+1} \ldots o^n$ ;
    $\mathcal{I} \leftarrow \mathcal{I} - R$ ;
$OI \leftarrow$ all overloaded instances with load larger than $\bar{l}_t$;
$UI \leftarrow \mathcal{I} - OI$;
/* Phase 2: identifying                                                */
foreach instance $o^j$ in $OI$ do
    $\theta \leftarrow \min\{l_{jt} - \bar{l}_t, \frac{u-v}{2}\}, S_k \leftarrow$ the substreams of $o^j$ ;
    while $S_k \neq \emptyset$ do
        $s_i \leftarrow$ get the largest substream such that $y_{it} < \theta$ ;
        insert $s_i$ into $PQ$;
        $l_{jt} \leftarrow l_{jt} - y_{it}, \theta \leftarrow \theta - y_{it}$ ;
        $S_k \leftarrow S_k - \{s_i\}$
foreach substream $s_i$ is assigned to an instance in $R$ do
    insert $s_i$ into $PQ$;
/* Phase 3: reassigning                                                */
while $PQ$ is not empty do
    $s_i \leftarrow$ peek the substream with the largest load from $PQ$ ;
    $o^j \leftarrow$ get the least-loaded instance from $UI$ ;
    $\mathcal{F}(s^i) \leftarrow o^j, l_{jt} \leftarrow l_{jt} + y_{it}$ ;
    if $l_{jt} \geq \frac{u+v}{2}$ then
        $UI \leftarrow UI - \{o^j\}, OI \leftarrow OI + \{o^j\}$ ;
return $\mathcal{F}$;
```

3 Eager Load Balancing

We now present the *eager load balancing* (ELB), which performs load balancing in each statistic window and leverages heuristics to reduce state movements as many as possible. In ELB, the objective of minimizing load imbalance is relaxed to a range $[v, u]$, where v and u are the lower and upper bounds of the load can be hold by each instance. For this relaxation, it is much easier to find a feasible assignment with less state movements. In addition some substreams are being hot spots at runtime, which have large volume of load and challenge the process of load balancing. Two heuristic rules are leveraged by ELB: (1) distribute the hot spots as evenly as possible; (2) fit the load of each instance into the range $[v, u]$ and make it as close as possible to $\frac{u+v}{2}$. Furthermore, we assume that the load for each substream in any window satisfy $y_{ik} \leq \frac{u-v}{2}$, which can be fulfilled by choosing suitable value for p and partition function.

The algorithm ELB, as depicted in Alg. 1, includes three phases. In the first phase, we first calculate the load vector $L_t = (l_{1t}, \ldots, l_{nt})$ according to the latest histogram Y_t and the current assignment $\mathcal{F}_1$. Let w be the overall load, $w = \sum_{j=1}^{n} l_{jt}$, then average load is $\bar{l} = \frac{w}{\pi}$, where π is the new parallelism such that $\pi = \lceil \frac{2w}{u+v} \rceil$. If $\pi > n$, then $\pi - n$ instances will be added into $\mathcal{I}$. Loads for these newly added instances are 0. If $\pi < n$, then $n - \pi$ instances should be removed from $\mathcal{I}$. To minimize state movements, we pick the $n - \pi$ least-loaded instances and keep them in a set R. The substreams assigned to instances in R should be reassigned to the instances in $\mathcal{I} - R$. All instances are sorted in a descending order of loads. We use two sets OI and UI to keep tracking the overloaded and underloaded instances respectively (Line 4–8). We only allow moving substreams from OI to UI, for which the state movements are reduced efficiently.

The purpose of the second phase is to identify the substreams that should be reassigned for each overloaded instances (Line 15–23). For an overloaded instance o^j, the load for the substreams that could be moved out is at most $l_{jt} - \bar{l}_t$. Since load for each substream is under $\frac{u-v}{2}$, the chosen substream must have its load under a threshold $\theta = \min\{l_{jt} - \bar{l}_t, \frac{u-v}{2}\}$. Each time we choose the largest substream of load smaller than θ (Line 18–20). The value of θ and load for o^j should be updated thereafter and then continue the search for the next substream. Repeat the process until no substream of o^j satisfying the condition (Line 17–21). The substreams assigned to o^j are also supposed to be sorted in a descending order of load, and hence the search completes in one traversal. Moreover, R is not empty if $\pi < n$, therefore the substreams processed by the instances in R should be reassigned to other instances outside (Line 22–23). All identified substreams are added into PQ.

In the last phase (Line 19–26), we reassign the chosen substreams to the underloaded instances in UI. The substreams in UI are listed in a descending order of loads. The process completes by repeating the *first-fit procedure*, where each time we choose a substream s_i with the largest load and assign it to the least-loaded instance $o^j \in UI$. If o^j is overloaded thereafter, then it will be removed from UI and added into OI.

4 Correlation-based Algorithm

In contrast to ELB, we execute a load balancing every m, $m > 1$, statistic windows. To reduce load imbalance, we compute an assignment that fits for a sequence of histograms $\boldsymbol{Y} = (Y_1, \ldots, Y_m)$ over m windows. The overhead of state movements is amortized over m windows and it is negligible if m is large enough. Therefore, we can ignore the overhead of state movement and only focus on minimizing the load imbalance.

We are given an assignment $\mathcal{F}$ and a sequence of load vectors $\boldsymbol{L} = \{L_1, \ldots, L_m\}$. Since $var(L_j) = \frac{1}{n}\sum_{i=1}^{n} l_{ij}^2 - \bar{l}_j^2$, the overall load imbalance can be written:

$$\sum_{j=1}^{m} var(L_j) = \sum_{j=1}^{m} \big(\frac{1}{n}\sum_{i=1}^{n} l_{ij}^2 - \bar{l}_j^2\big) = \frac{1}{n}\sum_{j=1}^{m}\sum_{i=1}^{n} l_{ij}^2 - \sum_{j=1}^{m} \bar{l}_j^2 \tag{3}$$

Each substream s_i associates with a *load series* $X_i = (y_{i1}, \ldots, y_{im})$. X_i can be viewed as a discrete-time stochastic process $X_i = \{y_{it} : t \in \mathbb{N}^+\}$, where y_{it} is the number of tuples of s_i arrived in the t-th window. Let $S_i = \{s_1, \ldots, s_r\}$ be the substreams that is assigned to instance o^i ($1 \leq i \leq n$), then $\mathcal{S} = \cup_{i=1}^{n} S_i$ and $S_i \cap S_z = \emptyset$ if $i \neq z$. Let $N_i = X_1 + \cdots + X_r$ and $\eta_i = E(N_i) = \sum_{s_i \in S_i}^{|S_i|} E(X_i)$, then we have

$$\sum_{i=1}^{n} var(N_i) = \frac{1}{m}\sum_{i=1}^{n}\sum_{j=1}^{m} l_{ij}^2 - \sum_{i=1}^{n} \eta_i^2. \tag{4}$$

Algorithm 2: Correlation-based Load Balancing (CLB)

```
Input: Load series $\{X_1, \ldots, X_p\}$
Output: Assignment $\{S_1, \ldots, S_n\}$
1  Initialization: $S_1 \leftarrow \{s_1, \ldots, s_p\}$, $r \leftarrow 1$ ;
2  foreach substream $s_i$ do
3      $\omega_i \leftarrow 0$ ;
4      foreach substream $s_j$ ($j \neq i$) do
5          $cov(X_i, X_j) \leftarrow E[X_i X_j] - E[X_i]E[X_j]$ ;
6          if $cov(X_i, X_k) \geq \theta$ then
7              $\omega_i \leftarrow \omega_i + cov(X_i, X_k)$ ;
8  while $r \leq n$ do
9      $s_k \leftarrow$ the substream with the maximum weight $\omega_k$ ;
10     $S_h \leftarrow$ get the set containing $s_k$ ;
11     foreach substream $s_i$ of $S_h$ do
12         if $cov(X_k, X_i) \leq \theta$ then
13             $S_h \leftarrow S_h - \{s_i\}$, $S_r \leftarrow S_r \cup \{s_i\}$ ;
14             $\omega_i \leftarrow 0$;
15             foreach substream $s_j \in S_h$ do
16                 $\omega_j \leftarrow \omega_j - cov(X_i, X_j)$
17             foreach substream $s_j \in S_r$ do
18                 $\omega_j \leftarrow \omega_j + cov(X_i, X_j)$
19     if $r < n$ then
20         $r \leftarrow r + 1$ ;
21         $S_r \leftarrow \emptyset$;
22     else
23         return; // already $n$ subsets
```

By some transformations of Eq. (3) and Eq. (4), we can prove that $\min \sum_{j=1}^{m} var(L_j)$ is equivalent to $\min \sum_{i=1}^{n} var(N_i)$. In addition, by studying the variances $var(N_k) = var(X_1 + \cdots + X_r)$ and $var(X) = var(X_1 + \cdots + X_p)$, we have

$$var(X) - \sum_{k=1}^{n} var(N_k) = 2 \sum_{X_i \in S_k, X_j \in S_z, k \neq z} cov(X_i, X_j) \qquad (5)$$

The right component $var(X) - \sum_{k=1}^{n} var(N_k)$ in Eq. (5) is denoted as *cross covariance*, which counts the covariances of substreams that fall into different subsets. Since $var(X)$ is a constant, minimizing $\hbar(\mathcal{F})$ is equivalent to finding a partition of $\mathcal{S}$ into subsets $S_1 \ldots S_n$ that maximize $var(X) - \sum_{k=1}^{n} var(N_k)$.

We construct a complete graph $G = (V, E)$ from the load series $X_1 \ldots X_p$, where a vertex $v_i \in V$ represents the load series X_i and each edge $e_{ij} \in E$ connecting v_i and v_j, $v_i, v_j \in V$, is assigned with weight $2cov(X_i, X_j)$. Let $n = 2$, then $\max[var(X) - \sum_{k=1}^{n} var(N_k)]$ is equivalent to computing the *Max-cut* for G. However, the *Max-cut* problem is NP-complete,and thus we present a greedy solution, as shown in Alg. 2, in which each time we choose a substream s_k based on an alternative metric and split the set containing it to two subsets.

Given a threshold θ, $0 \leq \theta < 1$, and a substream $s_1 \in S$, we consider a split of the set S into two subsets S_1 and S_2, where S_1 keeps s_1 and any substream s_i such that $cov(X_1, X_i) \geq \theta$ and S_2 includes others otherwise, i.e., $S_1 = \{s_1\} \cup \{x_i | x_i \in S, cov(X_1, X_i) < \theta\}$ and $S_2 = S - S_1$. Let ω_1 be the contribution of o_1 to the cross covariance in this split, then $\omega_1 = \sum_{s_i \in S, i \neq 1} cov(X_1, X_i)$, if $cov(X_1, X_i) \geq \theta$. Calculation of the covariance matrix $\Sigma = [cov(X_i, X_j)]_{p \times p}$ is described by Line 2–7.

The set splitting proceeds in runs (Line 8–23). For each run we choose the substream with the largest contribution to perform a set splitting rather than maximize the overall cross covariance, which is NP-complete as we showed earlier. Suppose that substream s_k has the largest weight in the current run, i.e., $s_k = \max\{w_i | s_i, i = 1 \ldots p\}$, then the set S_h containing it will be split into two subsets S_h and S_r, where S_r is an empty set and each substream s_i such that $cov(X_k, X_i) \geq \theta$ will be move to S_r from S_k. Since ω changes as set splitting, we should update its value for each substream of S_h and S_r to prepare the next run (Line 15–18). Repeat this procedure until n sets are created.

5 Evaluation

We evaluated our algorithms with three metrics: (1) *load imbalance* $var(L_t)$, (2) *state movements*, and (3) *processing latency*. The processing latency measures the time for processing each tuple. Based on this measurements, we can also calculate the system throughput $1/avg$, where avg is the average processing latency for a stream of tuples. In the experiments, we compared *ELB* and *CLB* with two existing solutions:

PKG also implements the key grouping but it was designed for stateless LB [10].

UHLB balances load with a universal hash function rather than the key grouping in our context. It returns $h(t)$, where $h : [p] \rightarrow [n]$ is chosen at random from a family of 2-universal hash functions.

Datasets. Two types of datasets, both real and synthetic, are used in this evaluation.

Twitter stream. The real dataset consists of a collection of tweets extracted from an interval around 29 hours: Feb 27 15:24:12—Feb 28 20:47:34, 2013. There are 10,637,691 tweets and about 13.9 GB in total. Each tweet is viewed as a tuple of JSON objects.

Synthetic stream. Two synthetic streams S1 and S2 are used to simulate the fluctuation of data rate and the change of data distribution respectively. S1 and S2 conform to a relational schema (ts, a_1, a_2), where ts is a Unix timestamp, a_1 is an integer falls into [1,100], and a_2 is a string of words. The field a_1 is designated as the partition key on which S1 and S2 have been partitioned into 100 substreams. The partition keys of S1 and S2 follow the Gaussian and Zipf distributions respectively, which are used to simulate various data skewness. The means for Gaussian and Zipf are set as the same.

In addition, a Poisson process is used to control the data rates of S1 and S2. In a Poisson process, tuples arrive sequentially and their inter-arrival times Z_i are exponentially distributed with a rate parameter λ : $Prob\{Z_m \leq \tau\} = 1 - e^{-\lambda\tau}$, where the parameter is $\lambda = 10000$.

5.1 Simulation Results

Experimental results are based on two hours simulation. In this experiment, we implement a simple topology, as shown in Fig. 1, where the operator u is responsible for generating tuples or read data from Amazon S3. The operator v serves as a sink for collecting the statistics for o. Operator o is used for evaluating the tested algorithms.

$$\begin{tikzcd} u \arrow[r] & o \arrow[r] & v \end{tikzcd}$$

Fig. 1: **A simple topology with 3 operators.** The size of state of o is set to one tenth of the data rate, $\psi(o) = \frac{1}{10}r(o)$.

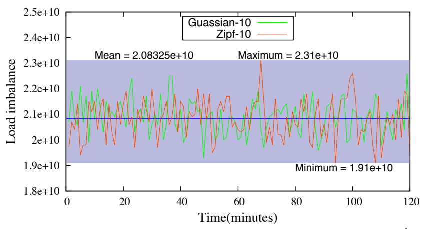

Fig. 2: Load imbalance with $\pi = 10$ and $^1/_f = 1$

The length of a statistic window is set to 1 minute and thus we have 120 histograms $\boldsymbol{Y} = (Y_1, \ldots, Y_{120})$ for each stream in total. To compare the load imbalance, the number of instances in the experiment is fixed.

Load imbalance with respect to data distribution— As we claimed earlier, load imbalance is mainly caused by skewness of data. Therefore, we use S1 and S2, have different distributions, to investigate the impact of data skewness. Since S1 and S2 follow the same traffic model, i.e., they have approximately the same data rates, the load imbalances are only determined by data skewness. Fig. 2 shows the change of imbalance over time for CLB when we use 10 instances, i.e., $n = 10$. The results are similar for $n = 5$ and $n = 15$. The experiments on other algorithms also show similar features, and thus we just take CLB as an example.

Let Y_i and $Y_i^{'}$ be the histograms for the i-th statistic windows of S1 and S2 respectively, where Y_i satisfies the normal distribution and $Y_i^{'}$ satisfies the Zipf distribution. The variance of $Y_i^{'}$ is larger than that of Y_i, although $Y_i^{'}$ and Y_i have equal means. The imbalances over statistic windows are plotted in Fig. 2, in which the parallelism n is 10 and $^1/_f = 1$. We calculate the mean and standard deviation of the imbalances. As we expected, the mean of $var(L_i), i = 1 \ldots 120$, is 2.08983e+10, which is approximately equals to the mean of $var(L_i^{'})$ (2.08325e+10). The standard deviations of $var(L_i)$ and $var(L_i^{'})$ are 6.76016e+08 and 7.92797e+08 respectively. Therefore, the fluctuation of $var(L_i)$ is much severer than that of $var(L_i^{'})$. This is confirmed by the plots in Fig. 2. The lines labeled "Gaussian-10" and "Zipf-10" in the figure capture the fluctuation of imbalances $var(L_i)$ and $var(L_i^{'})$ of CLB on S1 and S2 respectively. The maximum and minimum imbalances occur in the line labeled "Zipf-10". The range between the maximum and minimum values on "Zipf-10" is colored with blue. By looking at the figure, all points of $var(L_i)$ falls into the range colored with blue and thus the change of $var(L_i)$ is much more moderate. This confirms that the data skewness has significant impact to load imbalance.

Performance comparison of LB algorithms— We used the real dataset to test the performance on load imbalance and state movements for each algorithm. The results for $n = 10$ are plotted in Fig. 3 and Fig. 4. In general, as we expected, UHLB and PKG beat our algorithms on load imbalance, but they perform much worse on state movements. In terms of CLB, UHLB and PKG reduce imbalance by at least an order of magnitude. The reason is apparent that the primary objective of CLB is to minimize state movements rather than load imbalance.

We have calculated the standard deviation of the imbalance $var(L_t)$ for all algorithms. In the experiments, the frequency of load balancing is set to $^1/_f = 24$, i.e., there are 5 load balancing in total. Table 1 summarizes the mean μ and standard deviation δ

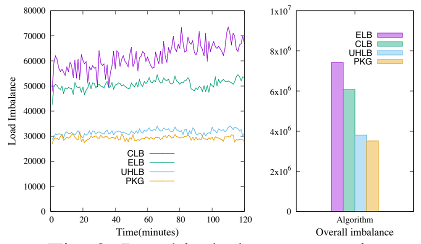

Fig. 3: Load imbalance over time.

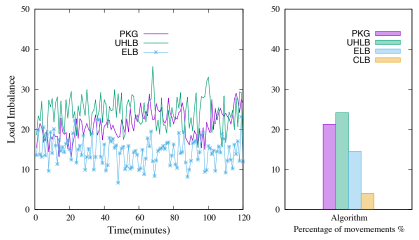

Fig. 4: Percentage of state movements.

Table 1: Mean and standard deviation

$1/f$		CLB	ELB	PKG	UHLB
1	μ	—	5.1E+4	3.1E+4	2.9E+4
	δ	—	1806.8	1018.8	770.4
24	μ	6.1E+4	—	—	—
	δ	5133.7	—	—	—

Table 2: Processing latencies(ms)

latency	CLB	ELB	PKG	UHLB
max	1103.13	1109.51	1551.30	1505.13
mean	0.76	0.73	0.92	1.01
median	0.30	0.33	0.38	0.38
95%	1.12	0.68	1.70	1.89

of imbalance $var(L_t)$ of all algorithms. The average percentage of state movements for CLB is 48.4% when $1/f = 24$. The value drops to 1.6% when we amortize them over the statistic windows.

By looking at Fig. 3, we can observe that ELB performs better than CLB on load imbalance, which is determined by their optimization objective and hence justifies the assertion we addressed earlier. CLB aims at minimizing the overall load imbalance $\hbar(\boldsymbol{L})$ by greedily reducing the covariance. In contrast, ELB executes load balancing eagerly at each statistic window. Fig. 4 shows the comparison of state movements. The left figure plots the percentage of movements for ELB, UHLB, and PKG. By looking at the figure, it is apparent ELB has much smaller state movement than UHLB and PKG. The average percentages of PKG, UHLB, and ELB is 21.2%, 24.2%, and 14.5% respectively. In the right figure, we compared the average percentages of PKG, UHLB, and ELB with the amortized percentage of CLB. As we expected the state movements of CLB is negligible comparing to the other three algorithms.

5.2 Processing Latency

We implemented the algorithms in Enorm [6,7], which extends Apache Storm [1] by integrating the ability of dynamic reconfiguration at runtime [8]. The experiments are conducted on Amazon's EC2 with medium VM instances, where each has 1.7 GB of RAM, moderate IO performance and 1 EC2 compute unit.We evaluated the metric by explicitly scaling out an operator *WordCounter* that counts the occurrence for each word every 1 minute over the Twitter stream. To exclude the interference from other factors, we fix the processing capacity of each VM to 1000 tuples/s. The data rate of Twitter stream starts at 1000 tuples/s and linearly grows to 16,000 tuples/s, and Storm scales out the operator by adding one more instances each time.

Processing latency with respect to data rate— Statistics of processing latency is illustrated in Tab. 2, where 95% is the 95-th percentile. By examining the 95-th percentile, we know that most tuples have processing latency less than 1.89 ms. In contrast, a very small portion of tuples have very high latencies. It confirms that state movement

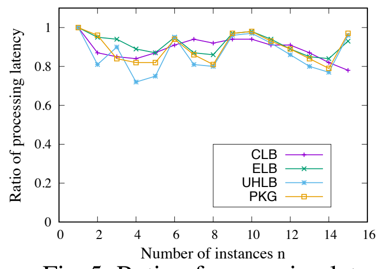

Fig. 5: Ratio of processing latency.

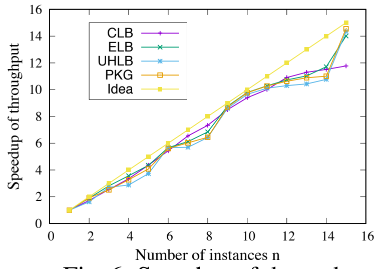

Fig. 6: Speedup of throughput.

indeed has significant impact to the processing latency of tuples. As we can see from the table, the maximum latency reaches up to 1.5 second. The processing latency is mainly due to stream buffering and replay. In the implementation of load balancing, we adopt a *pause-configuring-resume* procedure, and thus tuples from upstream operators will be buffered and then replayed to downstream after the completion of the process.

By comparing the mean of processing latency, we can assert that our algorithms outperform the existing solutions. In particular, CLB approaches the least reduction of 17% and ELB reduces the mean of processing latency up to 50%. To have better understanding of the processing latency, we calculated the ratio $\frac{\mu_1}{\mu_i}$, where μ_1 is the mean of processing latency of CLB when $n = 1$ and μ_2 is the mean of processing latency of any algorithm when $n = i$, $i = 1 \dots 15$. Fig. 5 plots the ratio by varying the number n of instances. It is apparent that UHLB and PKG fluctuate more severely than CLB and ELB.

Speedup of throughput—The speedup of throughput achieved by each algorithm is illustrated in Fig. 6. In the figure, the line labeled “Ideal” represents the theoretical speedup of scaling out the operator. The speedups for ELB and CLB are approximately linear to the parallelism. In contrast, PKG and UHLB cannot approach linear speedups. The change of speedups for the latter two algorithms show interesting features. By looking at the figure, we can observe remarkable phase transition on the lines labeled “PKG” and “UHLB”. The two lines can be divided into multiple stages, such as the ranges 3–5, 6–8, and 10–14. The speedup improves slightly in a stage, but it shows a sudden jump at the end of that stage. This phenomenon undoubtedly confirms the impact of load balancing. During the execution of a load balancing, the incoming tuples are temporarily buffered by the upstream operator. The buffered tuples would get congested if there are too many state movements involved in the load balancing. The upcoming tuples are delayed until the congested tuples have been processed and then we can observe a sudden jump of the speedup.

We also observe that the speedup of CLB gradually deviate from the “Ideal” line as we scale out the operator. As we can see from the figure, ELB, PKG and UHLB outperform CLB when $n = 15$. Since the execution of load balancing is infrequent, $1/f > 1$, for CLB, load imbalance cannot be removed in time. The overhead is too high for a single load balancing and this problem get worse when we have more instances. Consequently, the throughput declines seriously due to the load imbalance. It shows that the frequency f of load balancing is also very important to throughput. We have to carefully choose the value for f.

6 Conclusion and Future Work

We have shown that stateful load balancing for a streaming computation is a bi-objective optimization problem. It is NP-hard and we proposed two approximate algorithms, ELB and CLB, in which the objectives of minimizing load imbalance and state movements are relaxed. The evaluation shows that our approaches outperform the existing solutions in processing latency and throughput even though them have higher load imbalance. However, our approaches can be strengthened in a number of ways, which are deferred into the future work. For instance, the current algorithms are based on the histograms of historic data but we did not leverage them to examine the change of a stream, which could be very helpful for the decisions on load balancing.

References

1. Apache Storm. `http://storm.apache.org/`.
2. D. J. Abadi, Y. Ahmad, and et al. The Design of the Borealis Stream Processing Engine. In *CIDR'05*, Asilomar, CA, January 2005.
3. B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB journal*, 23(4):517–539, Aug. 2014.
4. B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *INFOCOM'04, Hong Kong, China, March 7-11, 2004*.
5. D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. VLDB '04, pages 180–191. VLDB Endowment, 2004.
6. K. G. S. Madsen, P. Thyssen, and Y. Zhou. Integrating fault-tolerance and elasticity in a distributed data stream processing system. SSDBM '14, New York, NY, USA, 2014. ACM.
7. K. G. S. Madsen and Y. Zhou. Demo: elastic mapreduce-style processing of fast data. In *DEBS '13*, pages 335–336, 2013.
8. K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. *CoRR*, abs/1602.03770, 2016.
9. K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 227–230, 2017.
10. M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE'15, Seoul, South Korea, April 13-17, 2015*, pages 137–148, 2015.
11. K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi. A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms. In *NCCA'12, London, United Kingdom, December 3-4, 2012*, pages 137–142, 2012.
12. S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. PACT '12, pages 53–64, New York, NY, USA, 2012. ACM.
13. M. A. Shah, S. Chandrasekaran, J. M. Hellerstein, J. M. Hellerstein, S. Ch, S. Ch, M. J. Franklin, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE'02*, pages 25–36, 2002.
14. S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? DEBS'12, pages 278–289, New York, NY, USA, 2012. ACM.
15. Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. VLDB '06, pages 775–786. VLDB Endowment.
16. Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. ICDE '05, pages 791–802. IEEE Computer Society, 2005.