2017 IEEE 33rd International Conference on Data Engineering

Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine

Kasper Grud Skat Madsen
University of Southern Denmark
kaspergsm@gmail.com

Yongluan Zhou
University of Southern Denmark
zhou@imada.sdu.dk

Jianneng Cao
Institute for Infocomm Research in Singapore
caojn@i2r.a-star.edu.sg

***Abstract*—Load balancing, operator instance collocations and horizontal scaling are critical issues in Parallel Stream Processing Engines to achieve low data processing latency, optimized cluster utilization and minimized communication cost respectively. In previous work, these issues are typically tackled separately and independently. We argue that these problems are tightly coupled in the sense that they all need to determine the allocations of workloads and migrate computational states at runtime. Optimizing them independently would result in suboptimal solutions. Therefore, in this paper, we investigate how these three issues can be modeled as one integrated optimization problem. In particular, we first consider jobs, where workload allocations have little effect on the communication cost, and model the problem of load balance as a Mixed-Integer Linear Program. Afterwards, we present an extended solution called ALBIC, which supports general jobs. We implement the proposed techniques on top of Apache Storm, an open-source Parallel Stream Processing Engine. The extensive experimental results over both synthetic and real datasets show that our techniques clearly outperform existing approaches.**

I. Introduction

Recently, Parallel Stream Processing Engines (PSPEs), such as Apache Storm [1] and Google MillWheel [2], have attracted significant attention. A job in an PSPE consists of a set of operators, each being parallelized into a number of instances, whose output would be fed into the downstream neighbors. All the operator instances of a job form a pipelined network, called the topology of the job.

There are three highly correlated job optimization problems in a PSPE. First of all, load balancing across the processing nodes is critical to maximizing cluster utilization and minimizing data processing latency especially during temporary load spikes. As jobs in PSPEs are long-standing, load rebalancing decisions have to be made continuously and periodically to maintain satisfactory system performance.

Secondly, neighboring operator instances in PSPEs continuously transfer data between each other. Collocating them at the same node can significantly reduce the system's workload by eliminating the overhead of cross-node data transmission, which includes both network bandwidth consumption [3], [4] and CPU consumption caused by data serialization and deserialization. To simplify the problem, previous studies of dynamic load balancing in PSPEs, such as [5]–[7] largely ignore the effect of collocating operator instances.

Thirdly, with the development of cloud computing platforms and resource management systems (such as Mesos [8]), runtime horizontal scaling of an PSPE, i.e. dynamically acquiring and relinquishing computing resources at runtime, is crucial to achieve high resource utilization and low operational cost. We argue that the problem of horizontal scaling is tightly coupled with both load balancing and operator instance collocation. For example, the overload of the system could possibly be rectified by collocating operator instances that have a high communication volume between each other to save the communication overhead instead of acquiring more resources. Another example is that horizontal scaling, load balancing and operator instance collocation all involve state migrations, which may incur significant overhead. Hence, optimizing them integratively would result in a solution with better load balancing and lower overhead. However, there is a lack of study on how to perform such an integrative optimization.

To fill the gap, we revisit the load balancing problem in PSPEs and propose a novel solution that also takes operator instance collocation and horizontal scaling into account. We first study the scenarios, where collocating operator instances has little effect. Such scenarios occur when each operator instance has to transfer data to a lot of neighbors evenly. We model this problem as a *Mixed-Integer Linear Program* (MILP). Then we extend our solution to cases, where collocating operator instances can significantly improve the system performance. The extended solution, called ALBIC (Autonomic Load Balancing with Integrated Collocation), dynamically constrains the MILP so that it gradually improves the collocation at runtime while ensuring a user-defined load-balance constraint is met.

II. System Model

Data Model. Input data of each operator is modelled as a number of continuous streams of tuples in the form of $\langle key, value, ts\rangle$, where key is used to partition the operator's input stream, $value$ is content of the tuple, and ts is its timestamp. Both key and $value$ are opaque to the system.

Query Model. A job corresponds to a set of user-defined continuous queries. The queries can be formulated as an operator network, which is a directed acyclic graph $\langle O, E\rangle$, where each vertex is an operator O_i and each edge is a stream, with its arrow representing the direction of data flow.

Execution Model. Each input tuple of O_i is associated with a key and these input keys are partitioned into a number of non-overlapping subsets, each is called a key group and denoted as g_k. The main assumption is that the processing of key groups is independent of one another. Each key group

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.81

Algorithm 1 Adaptation Algorithm

```
1: for each node n_i ∈ N that is set to be removed by the
   scaling algorithm in previous periods do
2:   if n_i.keygroups is empty then
3:     terminate node n_i
4:   end if
5: end for
6: plan ← keyGroupAlloc()
7: if Scaling(plan) then
8:   // Wait until new nodes are allocated
9:   plan ← keyGroupAlloc()
10: end if
11: apply(plan)
```

g_k must therefore have an independent processing state σ_k. A cluster has a set of nodes $N = \{n_1, \ldots, n_{|N|}\}$ and each node n_i processes a non-overlapping subset of key groups from any operator. If key groups g_k and g_l are both processed at node n_i, they are said to be collocated.

Statistics. The system maintains statistics on the usages of CPU, memory and network bandwidth, as well as the input and output data rates of processing each key group. Such statistics are collected and calculated over every period $P_{i \to j} : [T_i, T_j]$. The length of the period $T_j - T_i$ is a tunable parameter, which is called the *statistics period length* (SPL). A load value of a particular resource is a percentage point in the range $[0, 100]$. Based on the statistics, we detect the *bottleneck resource* of the computation, i.e. the one with the greatest total usage in the whole system. The load balancing objective will use the load values of the bottleneck resource. We define $gLoad_k$ and $load_i$ as the average load value of the bottleneck resource over the latest SPL of a key group g_k and a node n_i, respectively.

III. Integrative Reconfiguration

A. Integrative Adaptation Framework

The adaptation framework takes horizontal scaling, load balancing and operator instance collocation into account [9]. Algorithm 1 presents our simple yet effective framework. The adaptation is run periodically. In lines 1-5, it starts by checking all the nodes that are marked for removal in the previous adaptation periods, and if all the key groups have been moved out from these nodes, then they can be safely removed.

After that, the algorithm calculates a potential allocation plan for the operator instances (line 6), which factors in both load balancing and collocation of operator instances. This new allocation plan will not be deployed immediately, but will be used when making decisions about whether horizontal scaling is needed (line 7). This is important to avoid unnecessary or undesirable scaling because:

- A potential allocation optimizes load balancing that may solve the overloading problem of a processing node without scaling;
- Collocation of communicating operator instances may decrease the total load on the system so that scaling out can be avoided;
- (Undesirable) Scaling-in will not be done if it is impossible to balance the load well enough among the remaining nodes.

Furthermore, after making a scaling decision in line 7, we will redo the allocation planning algorithm again (line 9), which will make an integrative decision for scaling, balancing and collocation.

B. Key Group Allocation

1) LP Solver: The following solution can be applied to situations, where there is little opportunity to minimize communication cost by collocating key groups.

Metric. To measure load imbalance, we define a metric called *load distance*, which is the largest difference between the exhibited load of any node in the cluster and the average load in the cluster. We would like to find a load balancing solution that minimizes the load distance, because this is the state, where the system can tolerate maximum load fluctuations at any node, without exhibiting under- or overload.

Let B be the set of nodes that are marked for removal by the horizontal scaling algorithm and A be the rest of the nodes such that $N = A \cup B$. Then the average load, $mean$, is defined as $\lceil \frac{1}{|A|} \cdot \sum_{n_i \in N} load_i \rceil$, where $load_i$ is the load of n_i. The objective can be formally stated below:

Objective: *Minimize* $\max_{n_i \in A} |load_i - mean|$ *and* $\sum_{n_i \in B}(load_i)$ *s.t. the cost of migration* $\leq$ *maxMigrCost.*

We model the cost of migrating a keygroup g_k as $mc_k = \alpha \cdot |\sigma_k|$, where $|\sigma_k|$ is the size of the state of keygroup g_k and α is a constant, chosen such that mc_k is the time to serialize the state on a node with average load.

The minimization problem is NP-hard, because it is an instance of the *Multi-Resource Generalized Assignment Problem* [10]. To derive a solution, we model the minimization problem as a *Mixed-Integer Linear Program* as follows.

Variables. The current allocation of key groups to nodes are represented by a set of binary variables $q_{i,k}$ with a value of 1 if key group g_k is located at node n_i or 0 otherwise. The new allocation of key groups is denoted by the binary variables $x_{i,k}$, which is defined in a similar way as $q_{i,k}$. A node n_i is marked for removal if $kill_i = 1$. The variables d, d_u and d_l model the load deviations from $mean$.

Mixed-Integer Linear Program

$$\min \quad w_1 \cdot d - w_2(d_u + d_l)$$

s.t.

$$(1) \quad \forall_{g_k \in G} : \sum_{n_i \in N} x_{i,k} = 1$$

$$(2) \quad \sum_{n_i \in N} \sum_{g_k \in G} [(1 - q_{i,k}) \cdot x_{i,k} \cdot mc_k] \leq maxMigrCost$$

$$(3) \quad \forall_{n_i \in N} : \sum_{g_k \in G} (x_{i,k} \cdot gLoad_k) \leq mean + (d - d_u)$$

$$(4) \quad \forall_{n_i \in N \wedge kill_i = 0} : \sum_{g_k \in G} (x_{i,k} \cdot gLoad_k) \geq mean - (d - d_l)$$

$$(5) \quad mean - d \geq 0$$

Constraints. Constraint (1) ensures that each key group is allocated to exactly one node. Constraint (2) bounds the maximum cost of state migration. Constraint (3) bounds the

maximum load per node. Constraint (4) bounds the minimum load per node, and is effectively disabled for nodes marked for removal. Constraint (5) is needed to ensure that d does not exceed the lower bound of load.

Explanation. The MILP minimizes the load distance, with the help of three variables and two constraints. Constraints (3) and (4) define the maximum and minimum load of each node in the cluster, where the limits are influenced by the variables d, d_u and d_l, such that the difference between maximum and minimum load can be minimized by choosing appropriate values of the variables. The variable d represents the maximum load deviation from $mean$ in all the nodes in A. By minimizing the value of d, it is guaranteed that at least the upper or the lower bound of load on all nodes will be tight. In order to make both the upper and lower bounds tight, we introduce the variables $d_u, d_l \in \mathbb{R}$, such that d_u (or d_l) is chosen to be greater than zero to tighten the upper (or lower) bound. The solution thus depends on the variable d being minimized first and then secondarily d_u and d_l being maximized. To achieve this, the constants w_1 and w_2 in the objective function should be chosen such that $w_1 >> w_2$.

2) Autonomic Load Balancing with Integrated Collocation: Data communication between key groups on separate nodes consumes not only bandwidth, but also CPU, as it requires data serialization and deserialization. Collocating key groups having significant communication can therefore reduce the system load.

The above MILP cannot be simply extended to model the collocation of key groups, as detecting if two key groups are collocated needs a quadratic formulation. For efficiency reasons, we avoid quadratic models, as they are computationally too expensive to solve. Therefore we propose a heuristic algorithm to run on top of the MILP, called ALBIC (Autonomic Load Balancing with Integrated Collocation).

ALBIC dynamically measures the benefit of collocating each pair of key groups. For each invocation, ALBIC optimistically collocates one set of key groups with maximum benefit. During dynamic load balancing, the collocated key groups would be considered for migration as indivisible units. If the load of a set of collocated key groups becomes too large, it will be split into relatively even-sized partitions. ALBIC works in four steps, as explained next.

Step 1 - Calculate Scores. Let $out(g_i)$ be the total data rate sent from a key group g_i within the timespan SPL and let $out(g_i, g_j)$ be data rate sent from key group g_i to g_j. To decide if a keygroup pair g_i, g_j can contribute to the overall collocation, the value of $out(g_i, g_j)$ should exceed a threshold value $avg(g_i) \cdot sF$, where the $avg(g_i)$ is defined as the total number of tuples sent downstream from g_i divided by total number of downstream key groups and sF is a score factor.

Step 2 - Maintain Collocation. ALBIC merges all existing collocated key group pairs into a minimum number of sets, such that any pair of sets whose intersection is not empty, will be replaced by the union of those two sets. A set cannot be too large, as that hinders good load balancing and state migration. To overcome this problem, each set is split into a number of partitions, attempting to ensure that (1) the migration cost of a partition Pmc is less than *maxMigrCost* and (2) the maximum load PL of any partition is less than $maxPL$. ALBIC uses a graph model, where each key group is modelled as a vertex and each edge has $weight = out(g_i, g_j)$. The weight of the vertex of g_i is set as the mc_i, its migration cost, if $Pmc/maxMigrCost > PL/maxPL$. Otherwise it is set as $gLoad_i$, the load of g_i. Ties are broken randomly. Balanced graph partitioning [11] is then applied to generate balanced partitions with minimum inter-partition weighted edge-cuts.

Step 3 - Improve Collocation. Select one random key group pair g_i, g_j, which is not currently collocated, and which has the maximum value of $out(g_i, g_j)$. Let n_1 and n_2 be the node where g_i and g_j are currently allocated respectively. There are three cases:

Case 1: neither g_i or g_j is part of any set of collocated key groups. This case is handled by adding a constraint to the MILP to collocate g_i, g_j on the node n_1 or n_2 with the smallest load. *Case 2*: either g_i or g_j is part of a partition of collocated key groups. This case is handled by adding a constraint to the MILP to collocate g_i, g_j on the node, where the key group that is part of a set of collocated key groups is located. *Case 3*: g_i and g_j belong respective sets of collocated key groups. This case is handled similarly as Case 1.

Step 4 - Solving. The constrained MILP is solved and the load distance of the resulting allocation is calculated. If the load distance is larger than $maxLD$ (the user-defined maximum load distance), the problem is resolved by forming smaller (more) partitions by reducing the $maxPL$ (max partition load) parameter.

IV. Experiments

Hardwares. All our distributed experiments are conducted on Amazon EC2 with Storm. A cluster consists of one *master node*, a set of *input nodes* and a set of *worker nodes*. The master node has instance type $m1.medium$, and executes the Storm master and Zookeeper. The worker nodes have instance type $m1.medium$ and are used to execute the actual job logic. The input nodes have instance type $m3.xlarge$ and are used to produce input to the processing nodes.

Metrics. We use *load distance* to measure the load imbalance. As shown in previous work [12], [13], a more balanced load distribution can be more resilient to short-term load fluctuations by minimizing the chances of overloading and the queueing latency during load spikes. To verify the collocation of communicating key groups can reduce the system workload, we use the metric *load index* to measure how the system load is changed over time. It is defined as the current average system load divided by the average system load right after the initialization phase.

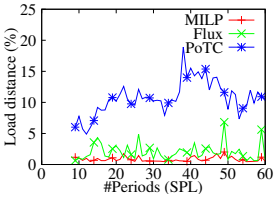

Fig. 1: Quality

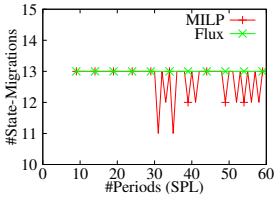

Fig. 2: #Migrations

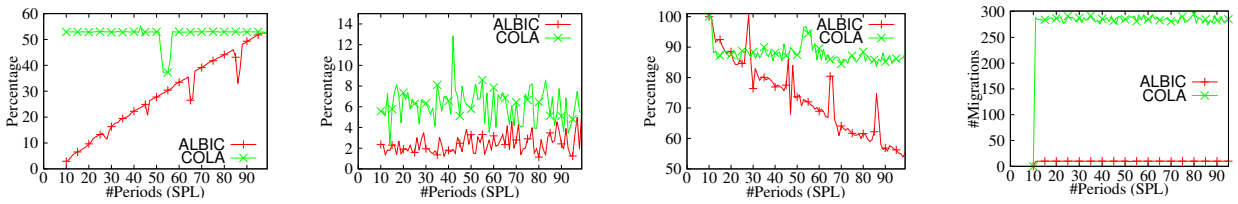

Fig. 3: ALBIC and COLA comparison

A. Load Balancing with MILP

The experiment investigates the quality and overhead of load balancing which can be obtained by solving the MILP. We first provide a comparison to Flux [5] and to the "Power of Both Choices" (PoTC) [7]. The experiment was executed on EC2, with one *input node* and twenty *worker nodes*. We use the dataset *Parsed Wikipedia edit history* [14].

The job consists of one input and three operators with one hundred key groups each. The first operator calculates a GeoHash value per input tuple. The second operator calculates TopK updated articles with a window of 1 minute and the last operator calculates global TopK updated articles, also with a window of 1 minute.

Figure 1 shows the load distances, directly after applying migrations. The MILP approach achieves a stable load distance consistently below 1%, while Flux exhibits significant fluctuations up to 7%. The reason that our approach outperforms Flux, is that Flux makes sub-optimal state migration decisions, i.e. Flux can require more state migration operations to load balance than what is necessary, meaning the *maxMigrations* constraint will then prevent Flux from achieving as good load balance as the MILP. PoTC cannot achieve a consistently good load distance. This happens because the job needs to do merge (every minute) and the amount of state to merge, varies over time and from node to node. This introduces skewness in the load, which is not considered by the PoTC approach. Notice also that the PoTC approach incurs a continuous overhead to process the merging operations, regardless of the load distribution.

B. Load Balance and Collocation (Real Data)

In this experiment, we compare ALBIC to COLA [15], which performs static optimization of both load balancing and collocation.

The job contains three operators, with five key groups per operator per node. The first operator extracts delays per airplane per year, and the second operator sums delays by airplane per year. The third operator sums delays per route, where a route is defined by having the same origin and destination airport. It is possible to obtain a perfect collocation of the first two operators, as they are parallelized similarly.

Figure 3 shows the collocation factor, load distance, load index and number of migrations done. As expected, COLA reaches the optimum collocation factor immediately because it optimize the plan from scratch. ALBIC, using its adaptive strategy, can gradually reach the same collocation factor over time. Furthermore, thanks to the MILP solver, ALBIC can consistently achieve a lower load distance than COLA, which only uses simple heuristics. The load index of ALBIC decreases from 100% to 50%, which means the system load has been cut in half due to the collocation of communicating key groups. COLA can only reduce the load by 25%, due to the large overhead of migrations. COLA migrates close to 300 key groups per SPL, while ALBIC only migrates 10 key groups.

V. Conclusion

We presented an integrated solution for load balancing, horizontal scaling and operator instance collocation suitable for general PSPEs. We first investigated how to model load balancing and dynamic scaling as an MILP, which is suitable when collocation of operator instances has little effect on the communication cost. As verified by our experiments, solving the MILP problem can achieve better load balance with a lower overhead of state migration compared to existing approaches. We then presented ALBIC, which optimizes the collocation of operator instances, while maintaining good load balancing and incurring low overhead at runtime. Our experiments verify that it maximizes the beneficial collocations without sacrificing the load balance and adaptation cost.

References

[1] A. Toshniwal *et al.*, "Storm@twitter," in *SIGMOD*, 2014.

[2] T. Akidau *et al.*, "Millwheel: Fault-tolerant stream processing at internet scale," in *VLDB*, 2013.

[3] P. Pietzuch *et al.*, "Network-aware operator placement for stream-processing systems," in *ICDE*, 2006.

[4] Y. Ahmad *et al.*, "Network awareness in internet-scale stream processing," *IEEE Data Eng. Bull.*, 2005.

[5] M. A. Shah *et al.*, "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*, 2003.

[6] Y. Zhou *et al.*, "Efficient dynamic operator placement in a locally distributed continuous query system," in *OTM*, 2006.

[7] M. A. U. Nasir *et al.*, "The power of both choices: Practical load balancing for distributed stream processing engines," in *ICDE*, 2015.

[8] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.

[9] K. G. S. Madsen, Y. Zhou, and J. Cao, "Integrative dynamic reconfiguration in a parallel stream processing engine," *ArXiV*, 2016.

[10] J. B. Mazzola *et al.*, "Heuristics for the multi-resource generalized assignment problem," *Naval Research Logistics*, 2001.

[11] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC*, 1998.

[12] Y. Xing *et al.*, "Providing resiliency to load variations in distributed stream processing," in *VLDB*, 2006.

[13] C. Lei and E. A. Rundensteiner, "Robust distributed query processing for streaming data," *ACM TODS*, 2014.

[14] "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.

[15] R. Khandekar *et al.*, "Cola: Optimizing stream processing applications via graph partitioning," in *Middleware*, 2009.