Dalton: Learned Partitioning for Distributed Data Streams

Eleni Zapridou
EPFL
eleni.zapridou@epfl.ch

Ioannis Mytilinis*
Oracle
ioannis.mytilinis@oracle.com

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

ABSTRACT

To sustain the input rate of high-throughput streams, modern stream processing systems rely on parallel execution. However, skewed data yield imbalanced load assignments and create stragglers that hinder scalability. Deciding on a static partitioning for a given set of "hot" keys is not sufficient as these keys are not known in advance, and even worse, the data distribution can change unpredictably. Existing algorithms either optimize for a specific distribution or, in order to adapt, assume a centralized partitioner that processes every incoming tuple and observes the whole workload. However, this is not realistic in a distributed environment, where multiple parallel upstream operators exist, as the centralized partitioner itself becomes the bottleneck and limits scalability.

In this work, we propose Dalton: a lightweight, adaptive, yet scalable partitioning operator that relies on reinforcement learning. By memoizing state and dynamically keeping track of recent experience, Dalton: i) adjusts its policy at runtime and quickly adapts to the workload, ii) avoids redundant computations and minimizes the per-tuple partitioning overhead, and iii) efficiently scales out to multiple instances that learn cooperatively and converge to a joint policy. Our experiments indicate that Dalton scales regardless of the input data distribution and sustains $1.3\times$ - $6.7\times$ higher throughput than existing approaches.

1 INTRODUCTION

Stream processing systems cope with data of enormous volume and velocity. From social network analytics to gaming, fraud detection, and stock trading, streaming applications require the real-time processing of high-throughput, in-motion data. Failing to sustain the input rate causes degradation in the quality of service and often jeopardizes the integrity of the entire application. To meet the ever-increasing computational demands, common wisdom suggests parallelization [3, 9, 20, 24, 29, 40, 43].
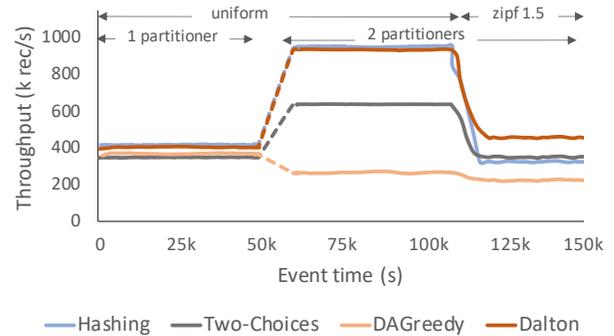
Figure 1: Impact of distribution changes and partitioner's parallelism on application's throughput

The physical limitations of a single machine and the inherently distributed nature of data sources (e.g., geo-distributed sensors in IoT) have sparked a lot of interest in distributed streaming frameworks such as Flink [20], Spark [43], and Kafka Streams [38], by both academia and industry. These systems compile the task at hand into a dataflow graph and follow a data-parallel approach, where different parts of the incoming data are assigned to different workers. Scaling the application under this model requires efficient *load balancing* – uneven assignments lead to stragglers and resource under-utilization. To make things worse, real data are often highly skewed, stressing the need for efficient parallelization [10, 28]. Therefore, a key research question is *how to partition streams in order to achieve balanced execution*. While shuffling trivially solves the problem for stateless operators, when state is involved, the optimal partitioning decision defines a complex optimization problem that is data-, resource- and workload-dependent.

As an example, consider a windowed group-by operation. To guarantee group-by semantics, a hash partitioner, which is sensitive to skewed data, is usually applied. To remedy this, previous research has proposed two techniques: i) *re-partitioning* [13, 15], and ii) *key-splitting* [30, 31]. Re-partitioning is too heavyweight as it involves state migration and transferring large data volumes over the network. By avoiding both the I/O cost of re-partitioning and the pitfalls of hashing, key-splitting has become the state-of-the-art. Key-splitting works in a Map-Reduce-like fashion. In the "map" stage, tuples are assigned to parallel workers. As each key can be assigned to multiple workers, key-grouping semantics are violated, but we benefit from the available parallelism even when the data distribution is heavily skewed. Then, data is partially aggregated and routed via hashing to the "reducers" for final aggregation. As Katsipoulakis et al. [21] have shown, key-splitting creates a trade-off between the effective parallelism in the first step and the aggregation cost in the second.

To identify the optimal trade-off, we have to answer many important questions such as: *which keys to split, how many workers do we need for each key, and which these workers should be.* To exacerbate things, the input rate and the underlying data distribution are not stationary but highly volatile and unpredictable: trending events create spikes in the load, and topic drifts change the set of "hot" keys that are responsible for load imbalance. Thus, key-splitting decisions should not be part of an offline, optimize-once process but a continuous and adaptive one.

During the past years, there have been many research efforts that employ key-splitting for stream partitioning under both the *tuple-at-a-time* [21, 30–32] and the *micro-batching* [1, 2] processing models. However, existing techniques suffer from at least one of the two following issues: i) they either do not adapt to distribution/rate changes, or ii) they cannot scale efficiently when the partitioner itself becomes the bottleneck and limits scalability. When the partitioner's input comes from multiple upstream operators, a single partitioning task may not be sufficient to sustain the load. Naively scaling by replicating the partitioner does not resolve the problem, as the locally-optimal decisions of each partitioner are not guaranteed to converge to a good global policy. Moreover, as existing partitioning functions are stateless, they: i) incur multiple redundant computations per tuple, further overloading the partitioner, and ii) miss the opportunity to exploit past experience for quickly converging to an efficient global policy.

Figure 1 illustrates a scenario that captures both deficiencies. An input stream with two parallel data sources produces uniform data, and all tuples go through a centralized partitioner. At $t = 50k$, we double the partitioners, and for the majority of the examined algorithms throughput increases, indicating that the partitioner itself had become the bottleneck. Then, after a while, one of the input streams becomes skewed due to a trending event. On the one hand, we observe that when following a static policy (Hashing, Two-Choices [31]), execution benefits from the second partitioner since individual partitioners follow the same strategy. However, static strategies cannot effectively handle all the different distributions. On the other hand, DAGreedy [32], which follows an adaptive policy, does not benefit from the second partitioner, as each of the two replicas acts independently and the system cannot converge.

This work proposes *Dalton*: a stream partitioning operator that can be injected into any stream processing system and jointly addresses both the adaptivity and the partitioner-scalability problem. To adapt to the distribution, Dalton relies on reinforcement learning (RL). For each assignment, a reward is instantly provided through a cost model that effectively captures distribution changes. To make our solution as lightweight as possible, we: i) reduce the RL state to the minimum, and ii) similarly to previous work [13, 15, 32], we employ a hybrid scheme that distinguishes heavy hitters from the tail of the distribution. Memoizing a state that keeps track of past experience reduces the per tuple overheads, as it avoids redundant computations and enables an efficient mechanism to scale the partitioners. By gathering the learned states of all partitioners, we can derive a new policy that is globally beneficial. Sharing experience and cooperatively learning a common policy happens through a distributed protocol with tunable synchronization overheads.

Our contributions are summarized as follows:

- We identify the joint *distribution shift - partitioner scalability* problem in stream partitioning and analyze its requirements and challenges.
- We propose Dalton: an RL-based stream partitioner that efficiently scales and maximizes throughput regardless of the data distribution. By adapting to the data and minimizing the per-tuple overheads, Dalton outperforms existing approaches by 1.3× - 6.7×.
- As centralized partitioners can become the bottleneck, we propose a distributed learning protocol that leverages locally learned states to converge to a common global policy. Our protocol achieves 1.4× to 3.4× higher throughput than simply replicating the partitioner.

2 PARTITIONING STATEFUL OPERATORS

The stream processing model assumes a dataflow, usually in the form of a directed acyclic graph, where nodes represent operators and edges data streams. We describe our formulation for the tuple-at-a-time processing model but extend it to the micro-batch model in Section 3.4. Each operator has an input and output queue and works at the tuple granularity, i.e., it pulls a tuple from the input queue, processes it individually, and enqueues it to the output.

Table 1: Notation table

S^w	stream of window w
$e_t = (t, k, \bullet)$	tuple with order t and key k
c_i , $1 \leq i \leq n$	partial aggregator subtasks
$P_t : S \rightarrow \{c_1, \ldots, c_n\}$	partitioning function
$L^{(t)}(c_i, w)$	load of c_i in w at time t
$I^{(t)}(P_t, w)$	load imbalance in w using P
$\Gamma^{(t)}(w)$	aggregation cost in w
$\mathbf{L}_w^{(t)}$	load vector of w
$\mathcal{X}_w^{(t)}$	fragmentation vector
$\mathcal{K}$	number of distinct keys

Streams & Windows. A stream S comprises an infinite sequence of records that obey a partial order. We consider the sliding window model (count- or time-based) and represent the records of a specific window w with S^w. We also assume that the order t of a tuple e in the stream is explicitly expressed as an attribute of the tuple, i.e., $e_t = (t, \bullet)$, and that it is used to assign tuples in windows.

Parallel dataflows. For each operator, distributed stream processing systems have multiple deployed instances that we call *subtasks*. For example, in Figure 2b, there are three subtasks for the parallel window aggregation. When exchanging data with downstream operators, the routing decision often depends on a set of specific attributes that act as partitioning keys. In the example of Figure 2b, tuples that have the same group-by key should be routed to the same subtask. To distinguish keys from the rest of the attributes, we denote tuples as $e_t = (t, k, v)$, where k is the key. In such key-based partitioning schemes, data skew causes significant performance degradation, as it decreases the effective parallelism. On top of that, the "hot" keys may change over time in an unpredictable manner. Re-partitioning requires re-distributing the state of stateful operators and incurs high I/O costs.
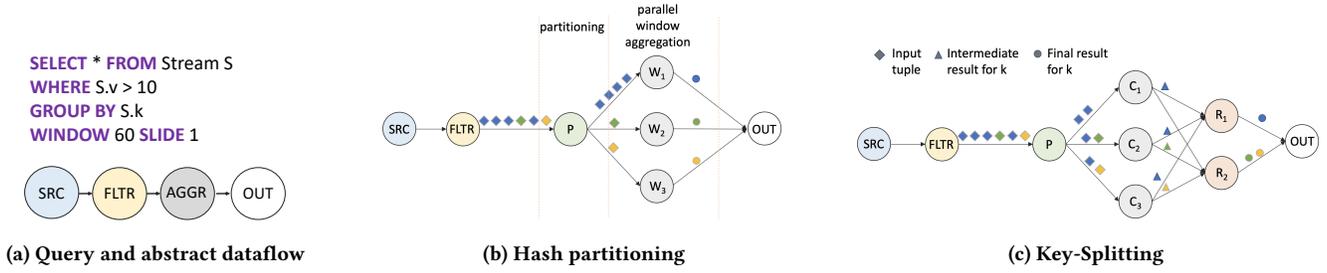
Figure 2: Example of partitioning in a dataflow with windowed aggregation

(a) Query and abstract dataflow   (b) Hash partitioning   (c) Key-Splitting

Key-splitting: adaptivity without re-partitioning costs. Key-splitting is a technique that allows dealing with diverse distributions without re-partitioning the state. The idea is to decompose the parallel stateful operators in two layers: In the first layer, we apply a partitioning function of the form: $P_t : S \rightarrow \{c_1, \ldots, c_n\}$, where c_i the partial aggregator subtasks[1]. Then, each subtask of the first layer assigns the partitioned tuples $S_i, i = 1, \ldots, n$ to windows and computes a partial aggregate for each S_i^w. In the next step, partial aggregates are routed via hashing to the second layer of subtasks for final aggregation. For simplicity, and as this model resembles Map-Reduce, we call the partial aggregators of the first layer *combiners* and the final aggregators *reducers*. Figure 2c demonstrates key-splitting for our group-by example. The Partitioner P uses an arbitrary scheme to assign tuples to combiners c_1, c_2, c_3, where tuples are partially aggregated, and hashing is used to route the partial aggregates to reducers r_1, r_2, where the final aggregates for each window are computed.

Challenge 1: Balanced work in combiners. Key-splitting permits arbitrary partitioning to combiners. However, to maximize the benefit from the available parallelism, we should minimize load imbalance. Assuming $L^{(t)}(c_i, w)$ denotes the number of tuples that are assigned to w and have been routed to c_i before the arrival of the tuple (t, k, v), and n the number of combiners, load imbalance is defined as:

$$I^{(t)}(P_t, w) = \left| \max_{1 \leq i \leq n} \{L^{(t)}(c_i, w)\} - \frac{1}{n} \sum_{i=1}^{n} L^{(t)}(c_i, w) \right| \quad (1)$$

Challenge 2: Minimal aggregation in reducers. Balancing the load of the combiners is not enough to achieve high end-to-end throughput, as the bottleneck can shift to the aggregation of the reducers. Thus, the second goal a good partitioning scheme should achieve is to minimize:

$$\Gamma^{(t)}(w) = \max_{1 \leq j \leq m} A_j^{(t)}(w), j = 1 \ldots, m \quad (2)$$

where m the number of reducers, and $A_j^{(t)}$ the cost of the j-th parallel reducer in the window w before the arrival of (t, k, v).

Challenge 3: Lightweight partitioners. Addressing both of the aforementioned challenges requires solving a multi-objective

optimization problem. However, as this problem is proven to be intractable [2], such an algorithm would require many computations per tuple. To meet the latency requirements of streaming applications, the partitioner should be lightweight and never become the performance bottleneck.

Challenge 4: Scaling the number of partitioners. Even if Challenge 3 is resolved and a really efficient partitioner is available, a partitioner that receives data at high rates from multiple parallel upstream operators/sources can still become the bottleneck. Scaling the number of partitioners is not trivial as, by default, partitioners do not communicate with each other, and local decisions may lead to a highly sub-optimal global partitioning. This is especially true when the data distribution from each upstream operator differs.

Assuming that: i) tuples never arrive out-of-order, ii) the content of a tuple does not affect the processing cost, and iii) hashing is used for routing to the reducers, we compile all four challenges in the *scalable and adaptive partitioning problem*:

PROBLEM 1 (SCALABLE & ADAPTIVE PARTITIONING). *Devise a partitioning scheme that distributes the load to the combiners and satisfies the following requirements:*

(1) It minimizes both $I(P_t, w)$ and Γ at the same time.
(2) It requires minimal latency per tuple.
(3) It quickly adapts to distribution changes.
(4) It can scale to multiple parallel partitioners.

State-of-the-art. While there is a lot of research on stream partitioning, no existing technique covers all four points. Most algorithms, e.g., Two-Choices [31], make static decisions that offer different imbalance-aggregation trade-offs but do not adapt at runtime [21, 30]. A state-of-the-art tuple-at-a-time algorithm of key-splitting is DAGreedy [32]. DAGreedy does adapt, but for each tuple, it calculates a score for each candidate combiner, and thus, the partitioning overhead increases with the number of parallel workers. Similarly, a state-of-the-art algorithm for the micro-batch model is Prompt [2], which also adjusts its strategy but has the overhead of sorting all keys in a batch based on their frequency. More importantly, both algorithms cannot efficiently scale out. In cases where multiple partitioning instances are deployed, as each of them optimizes only the locally observed distribution, partitioning policies diverge and degrade the overall system's performance. Even if the partitioners were syncing and were periodically exchanging load information, as the algorithm does not maintain state enriched with past experience, decisions between the sync points would again diverge, and convergence would never be achieved.

---

[1]The subscript in the partitioning function's symbol denotes that partitioning decisions are not fixed but can vary with time. At the same time, we show that changes in the partitioning policy are not necessarily coupled with the window specification.

3 LEARNING PARTITIONING POLICIES

The ever-changing nature of data streams makes static heuristics incapable to provide an efficient partitioning policy during the lifespan of a streaming application. Moreover, as we explained, techniques that rely on stateless partitioning functions, that forget past experience, increase processing per tuple and fail to scale in distributed environments. Reinforcement learning (RL) naturally fits this problem: it learns actions based on the actual data distribution, and as we show in Section 4, by keeping track of past experience, it enables a mechanism for scaling the partitioners. However, trivially applying RL results in a vast and impractical state-action space. In Section 3.1, we analyze the complexity of an RL-based solution and present three key-technical ideas that can decouple inter-dependent components of the problem and render it in a manageable form.

3.1 Cost of RL-based Stream Partitioning

We mathematically model the problem as a *Markov Decision Process* (MDP). Formally, an MDP is defined as a tuple $M = (\mathbf{S}, \mathbf{A}, P_a, R_a)$, where $\mathbf{S}$ is a finite set of states, $\mathbf{A}$ is a set of actions, P_a is the transition function that expresses the dynamics of the environment and R_a the reward function. More specifically, $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ denotes the probability that an action a taken at time t in state s, will lead to state s' at time $t+1$, and $R_a(s, s')$ the corresponding immediate reward. We consider the stream partitioner as an agent that takes actions and transitions across states with the aim of maximizing its cumulative reward. The environment is non-stationary, and each partitioning decision changes the load distribution in the combiners, thereby affecting future partitioning decisions. Next, we formally define states, actions, and rewards.

States. Given an input tuple $e_t = (t, k, v)$ and a window w, the state of the partitioner should capture the key attribute of the tuple at hand, and the current load distribution for the tuples in w that arrived before e_t, i.e., $\{e_{t'} : t' \in S^w \wedge t' < t\}$. To describe the load distribution, we use a *load vector* and a *fragmentation vector*.

Definition 3.1 (Load Vector). The load vector $\mathbf{L}_w^{(t)}$ is a vector that contains the number of tuples $L^{(t)}(c_i, w)$ that each combiner c_i received in the window w and before tuple e_t arrives, i.e.,

$$\mathbf{L}_w^{(t)} = [L^{(t)}(c_1, w), \ldots, L^{(t)}(c_n, w)]$$

Definition 3.2 (Fragmentation Vector). The fragmentation vector is defined as:

$$\mathcal{X}_w^{(t)} = [\mathbb{1}(k_1, w, t)_1, \ldots, \mathbb{1}(k_1, w, t)_n, \ldots, \\ \mathbb{1}(k_{\mathcal{K}}, w, t)_1, \ldots, \mathbb{1}(k_{\mathcal{K}}, w, t)_n]$$

where $\mathcal{K}$ the number of distinct keys in the window w. Conceptually, $\mathcal{X}_w^{(t)}$ is a bit-vector that, for each key k, shows which combiners hold at least one tuple corresponding to key k with order $t' < t$, in the window w.

Thus, we represent the state as a $(k, \mathcal{X}_w^{(t)}, \mathbf{L}_w^{(t)})$ triplet. Assuming $\mathcal{L}$ tuples within a window before e_t arrives, and following a "balls into bins" argument for the load, the number of possible states is:

$$\mathcal{K} \times 2^{\mathcal{K}n} \times \binom{\mathcal{L} + n - 1}{n - 1}$$

Actions. At each step, an action consists of the selection of a combiner for a given tuple e_t. Therefore, the number of available actions $|\mathbf{A}|$ corresponds to the number of combiners n.

Rewards. Based on Equations 1 and 2, the cost for an action e_t consists of the action's contribution to the: (i) imbalance in the combiners, and (ii) the reducers' aggregation cost. The cost can be translated to the following rewards function:

$$R_w(e_t, a) = -(p_1 * CI_w^{(t)}(a) + p_2 * CA_w^{(t)}(k, w))$$

where p_1, and p_2 are adjustable and control the contribution of each metric (i.e., $p_1 + p_2 = 1$). We express the first term as:

$$CI_w^{(t)}(a) = \frac{L^{(t+1)}(a, w) - \bar{L}_w^{(t+1)}}{max\{L^{(t+1)}(a, w), \bar{L}_w^{(t+1)}\}} \tag{3}$$

where $\bar{L}_w^{(t)} := \frac{1}{n}\sum_{i=1}^{n} L^{(t)}(c_i, w)$ denotes the average load of the combiners in the window w before the arrival of tuple e_t, and a refers to the chosen combiner. CI captures the cost of assigning one more record to the combiner a. The metric is normalized, taking values in the range $[-1, 1]$. Assigning the tuple to an underloaded combiner results in a negative CI. This corresponds to a high reward, encouraging such choices. Conversely, choosing an overloaded combiner is penalized with a low reward.

For the second term of the cost, we assume that the aggregation cost that action a incurs for the input tuple (t, k, v) is proportional to the fragmentation $||\mathcal{X}_w^{(t+1)}(k)||$ of key k, where $|| \bullet (k)||$ denotes the number of 1s in the $\mathcal{X}_w$ bit-vector that correspond to key k, and thus to how many combiners k is split. Again, we normalize the cost, which is expressed as:

$$CA_w^{(t)}(k) = \frac{||\mathcal{X}_w^{(t+1)}(k)||}{n} \tag{4}$$

Solving the above RL problem with a technique such as *Q-learning* [42] or *Sarsa* [36] would require tabulating $|\mathbf{S} \times \mathbf{A}|$ elements. Even for a toy example, with 10 distinct keys and 100 tuples already in the window, running with 8 combiners, the number of possible states is approximately 3×10^{35}. Furthermore, with each new tuple in this window, as the load increases, the number of possible states increases as well. The complexity is already prohibitive, and in reality, we have millions of tuples per window and hundreds of parallel worker threads, so this number will grow exponentially.

An offline learning approach, as in [1, 33] is also not going to work since such a train-once process violates the third requirement of Problem 1 – we need a partitioner that continuously learns in an online fashion and adapts to the data.

To decrease the number of states, we employ three key ideas:

Key idea 1: Separation of concerns Following prior art, we can decrease $\mathcal{K}$ by employing RL for the partitioning of the most frequent keys and hashing for the rest. As we show later in Theorem 3.4, the threshold we use to make this distinction results in a maximum of n frequent keys.

Key idea 2: Load space quantization. The number of possible values that the load vector can assume is the largest factor in determining the size of the state space, $|\mathbf{S}|$. To tame it, we can make the load assignment representation more coarse-grained by quantization. For a quantum q, assuming that a combiner c_i has a current load of $L(c_i, w)$, we transition to a new load value only when q more
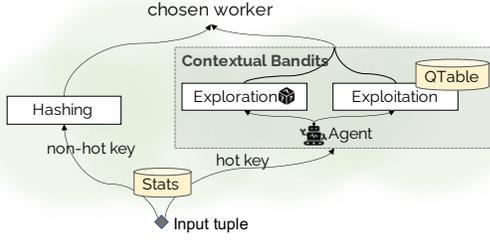
Figure 3: Dalton's workflow

tuples are assigned to c_i. The quantized representation can take one of $\binom{\frac{\mathcal{L}}{q}+n-1}{n-1}$ values, which corresponds to a significant reduction of the state space. With these two modifications, for $q = 10$ and considering 8 frequent keys, the number of state-action pairs in our example is already decreased to approximately 3×10^{24}.

Key idea 3: Temporal invariability of fragmentation. The new state to which we transition depends not only on the chosen action but also on the next tuple of the stream. This makes it infeasible to visit all possible states – the truly eligible states for a transition are conditioned on the order in which the keys appear. Let us assume that we are in state $(k_j, \mathcal{X}_w^{(t)}, \mathbf{L}_w^{(t)})$ and all combiners have loads corresponding to a new quantum. If the partitioner does not decide on a further split for the key of the incoming tuple (something it tries to avoid), then, for the given window w, $\mathcal{X}_w$ will not change as well. Hence, given these assumptions, the agent's actions contribute to the selection of the next state only every $q \leq T \leq n(q-1) + 1$ tuples. In between, the next state is solely determined by the keys in the stream. If we increase q a lot, in order to reduce the state space, then T will increase as well and the RL agent will degenerate to a contextual bandit [8, 26].

Algorithm 1: Dalton

---
   local : n: number of combiners
   input : Incoming tuple $e : (t, k, v)$
1 UpdateFrequency(k) ;
2 $f_k \leftarrow$ *EstimateFrequency*(k) ;
3 **if** $f_k \geq \frac{\mathcal{L}}{n}$ *or* (k *in* Q *and not expired*) **then**
4   assign e to $c^* = argmax_i\{Q(k, i)\}$;
5   compute reward $R(k, c^*)$;
6     $Q(k, c^*) = Q(k, c^*) + \gamma[R(k, c^*) - Q(k, c^*)]$;
7 **else**
8   assign e to $c^* = hash(k)$;
9 UpdateWorkerLoad(c^*) ;

3.2 Reducing State using Contextual Bandits

Contextual bandits can be used to learn a different policy per key – allowing for a key to be split according to its frequency – with considerably reduced memory requirements. Furthermore, by discounting past rewards, contextual bandits can be robust to distribution shifts and quickly adapt their policy in an online manner.

A contextual bandit maintains a *Q-table* per key and aims to learn an estimate of the value $Q(k, a)$ for all the possible assignments of key k to a combiner a. When presented with a new tuple, the partitioner selects the action (combiner) that maximizes the expected reward. A natural way to estimate $Q(k, a)$ is by averaging the rewards that have been received when the combiner a was selected for key k. Nevertheless, since streams are unpredictable and the underlying data distribution may change, the reward distribution is non-stationary and we may desire to rely more heavily on recent rewards than long-pasting ones. Let us denote with $Q_t(k, a)$ the estimated average reward for action a after observing the first $t - 1$ rewards. Then, given the t-th reward $R_t(k, a)$ for that action, we update the learned value with the following rule:

$$Q_{t+1}(k, a) = Q_t(k, a) + \gamma[R_t(k, a) - Q_t(k, a)] \qquad (5)$$

where γ is a constant step-size parameter that takes values in the range $(0, 1]$. Each key corresponds to a row in the Q-table and based on *Key-idea 1*, there can be at most n ("hot") keys. Each row in the Q-table has n entries – one for each possible action – which makes the total memory complexity of the algorithm $O(n^2)$.

Initial Values. We set the initial values to the minimum possible reward, i.e., -2 (Equations 3 and 4). This provides two nice properties that prevent excessive key splitting. First, after the initial assignment of a key k to a combiner c_i, subsequent records with key k have an affinity for the same worker; splitting happens only through exploration. Second, even when exploration splits a key and assigns it to a combiner c_j, due to the low initial value estimates, the partitioner will be discouraged from sending more records to c_j unless the reward is substantially higher. Without substantially higher rewards, when the tuple that the exploration assigned to c_j expires, the fragmentation of k will be decreased.

Exploration. The agent uses an ϵ-greedy policy: with a probability $1-\epsilon$ it greedily chooses the action with the highest $Q(k, a)$ value, and with a probability ϵ it explores new assignments by randomly choosing among all actions. This policy allows the partitioner to explore new assignments by splitting or even migrating a key. The probability ϵ should be low so that most of the time, the agent makes decisions that have been already proven to be beneficial. Our evaluation indicates that a good value is $\epsilon = 0.1$.

Heavy hitters. As already mentioned, we employ the contextual bandit agent for partitioning the most frequent keys and hashing for the rest. The intuition behind our definition for heavy hitters is that (i) the bandit should be used only for keys for which splitting can be beneficial and (ii) splitting is beneficial when a key causes imbalance even when it is the only heavy hitter assigned to a specific combiner. This idea leads to the following definition:

Definition 3.3 (Heavy Hitters). Heavy hitter is a key k whose frequency $f(k, w)$ within the window w satisfies $f(k, w) \geq \frac{\mathcal{L}}{n}$, where $\mathcal{L}$ the total load of the current window.

THEOREM 3.4. *There can be at most n heavy hitters in a window, where n is the number of combiners.*

PROOF. Let us consider that there are x heavy hitters $\{k_1, \ldots, k_x\}$. Then, according to Definition 3.3: $\sum_{i=1}^{x} f(k_i, w) \geq \frac{x}{n} \cdot \mathcal{L}$. But $\mathcal{L} = \sum_{i=1}^{x} f(k_i, w) + Y$, where Y is the total frequency of the non-heavy hitters. Thus, $x \leq n \cdot \frac{\sum_{i=1}^{x} f(k_i, w)}{\sum_{i=1}^{x} f(k_i, w)+Y} \leq n$ □
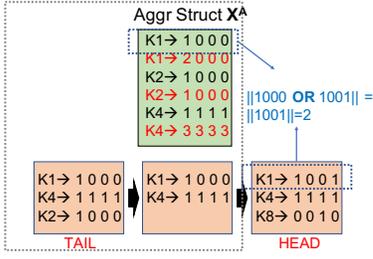
Figure 4: Data structures for maintaining $\mathcal{X}_w$

The problem with this definition is that the total load $\mathcal{L}$ of the current window is not known before the window completes. We solve this problem by using statistics from both the previous and the current window. Details follow in the next subsection.

We call our partitioning operator *Dalton* and present the pseudocode of the bandit algorithm it employs in Algorithm 1. Figure 3 presents an overview of Dalton's workflow. By maximizing the reward function, Dalton learns a policy that minimizes the imbalance and the aggregation cost and quickly adapts to distribution changes. This covers the requirements (1) and (3) of Problem 1. Next, we show the necessary enhancements to meet objectives (2) and (4).

3.3 Managing Windows

In the above discussion, we show that the computation of $\mathcal{X}_w$, $\mathbf{L}_w$, $R(c_i, w)$ and $f(k_j, w)$ depends on a window. This window is not necessarily the same for all four quantities. Here, we analyze the requirements for each of them and present the system design we use in order to achieve low latency in windowing operations and meet the second objective of Problem 1.

Let us assume an application with a sliding window of size W and slide s. In this case and for a starting point t_0, every s "time" steps[2], $(t_0, t_0 + s, \dots)$, each combiner emits a partial aggregate for the last window $((t_0 - W, t0], (t_0 + s - W, t_0 + S], \dots)$. As partitioning decisions must reflect the actual processing cost in both combiners and reducers, the estimated cost of an action must account for the load and fragmentation in $((t_0 - W, t0], (t_0 + s - W, t_0 + S], \dots)$. Therefore, the window we use for three out of the four quantities, $\mathcal{X}_w$, $\mathbf{L}_w$, $R(c_i, w)$, is W and it is updated every s steps.

Reward Computation. To compute the reward R in a sliding-window fashion, we need sliding-window data structures for $\mathcal{X}_w$ and $\mathbf{L}_w$. We opt for a design that has minimal update time and avoids costly memory allocations in the critical path. In an abstract level, both $\mathcal{X}_w$ and $\mathbf{L}_w$ follow a similar design: each has a dedicated memory pool of size $\lceil \frac{W}{s} \rceil$ that contains one pre-allocated block per slide organized in a circular linked list, and an extra structure that holds aggregated information. Using this design, each incoming tuple requires $O(1)$ update time by solely updating the head of the list. Slide expiration also requires an $O(1)$ update time to touch the tail of the list and the aggregate structure.

More specifically, for the fragmentation vector $\mathcal{X}_w$, at each slide, we get a block from the corresponding memory pool, and we maintain a map from the keys that appear in this slide to a bit-vector that indicates to which combiners the specific key has been assigned

$(k \to [\mathbb{1}_1, \dots, \mathbb{1}_n])$. Each time a tuple with key k_i gets assigned to a combiner c_j, we retrieve the map that exists in the head of the list/pool, get the bit-vector of k_i, and set the c_j-th bit to 1.

Assuming $\lceil \frac{W}{s} \rceil$ maps in the pool: $M_1, M_2, \dots, M_{\lceil \frac{W}{s} \rceil}$, where M_1 the head, the aggregate data structure $\mathcal{X}^{\mathcal{A}}$ maintains, in an incremental way, the union for all past slides, i.e., $M_2 \cup \dots \cup M_{\lceil \frac{W}{s} \rceil}$ and a reference counter per key per combiner that shows in how many of the past slides within the window, the key has been assigned to the combiner. Then, each time a slide expires, we do the following:

(1) Remove the tail of the list that corresponds to the expired slide and expire the corresponding keys. This consists of reducing the reference counter in $\mathcal{X}^{\mathcal{A}}$ that corresponds to each expired assignment and, if the counter becomes 0, setting the corresponding bit in the bit-vector of $\mathcal{X}^{\mathcal{A}}$.

(2) Merge the current head to $\mathcal{X}^{\mathcal{A}}$, by computing $\mathcal{X}^{\mathcal{A}} \cup M_1$, and increasing the corresponding reference counters.

(3) Use the expired memory block as the new head.

Assuming $\mathcal{K}_{HEAD}, \mathcal{K}_{TAIL}$ denote the key cardinality in the head and tail of the linked list, maintaining the $\mathcal{X}^{\mathcal{A}}$ structure incurs a cost of $O(\mathcal{K}_{HEAD} + \mathcal{K}_{TAIL})$ each time a slide expires, but allows the computation of Equation 4 in $O(1)$ time by simply computing the OR function between two bit-vectors: one retrieved from the head ($M_1(k)$), and one from $\mathcal{X}^{\mathcal{A}}(k)$ (Figure 4).

In a similar spirit, we compute the load of each combiner $L(c_i, W)$. Concretely, for every slide, we keep a counter that stores the number of tuples assigned to this combiner. Additionally, we maintain a sliding-window sum, corresponding to the total load of the combiner for all past slides, using the Subtract-on-Evict algorithm [37].

Statistics Computation. Although we need the window specification of the application (W, s) for computing the rewards, this is not true for the key frequencies $f(k, w)$. This window does not interfere with the application's semantics and is just used to identify the heavy hitters as time passes and distribution shifts. For the windowing in statistics updates, we use a tumbling window whose size is defined by the STATS_WIN system parameter; this is a tuning knob that affects the partitioner's latency. Then, heavy hitters are computed by using the formula of Definition 3.3. We estimate the load $\mathcal{L}$ of the current window by setting it equal to the load observed during the previous STATS_WIN window. Once a key is considered as a heavy hitter, it will be assigned using the bandits policy for the current and the next STATS_WIN window. At the end of the next window, if the key has not exceeded the frequency threshold again, it is expired and its entry is deleted from the Q-table. This allows the system to use past observations and continuously learn the assignment policy of keys that remain hot for more than one STATS_WIN window instead of resetting the Q-table at the end of every window.

Intuitively, when STATS_WIN is too small, it is like we "zoom in" a lot to the distribution and miss heavy hitters. Then, for the missed heavy hitters, hashing is used instead of the bandit policy, and thus, we allow for combiners to become stragglers and stale execution. At the other extreme, if STATS_WIN is too large, we approximate the distribution better, but in case of distribution shifts, we force unnecessarily many tuples to go through the bandit and incur extra performance overheads. For example, consider that the distribution

---

[2]"time" just expresses an ordering and refers to both count- and time-based windows

changes every T_1 sec and STATS_WIN = $2T_1$. Let us also assume that initially, we had a set of heavy hitters $\mathbb{K}_1$, and when the distribution changed at T_1, the new heavy hitters became $\mathbb{K}_2 : \mathbb{K}_1 \cap \mathbb{K}_2 = \emptyset$. Thus now, there are $|\mathbb{K}_1| + |\mathbb{K}_2|$ keys that go through the bandit, instead of just $|\mathbb{K}_2|$.

3.4 Dalton for Micro-batches

To increase throughput at the cost of latency, micro-batch systems accumulate incoming tuples and process them in batches. Each operator pulls a batch from its input queue, processes it individually, and enqueues it to the output. Thus, the partitioner is expected to first *see* all the tuples of a batch, split them into subsets called *data blocks*, and emit each data block to a combiner. Observing all tuples of a batch before taking decisions can lead to more accurate statistics and thus assist the whole partitioning process.

Typically, to perform windowed computations, a partial aggregate is first calculated for every data block, followed by a final aggregation that combines intermediate results (as in Section 2). However, combiners do not reduce data at the window but at the micro-batch level. Hence, the basic implementation difference is that we must modify fragmentation vectors $\mathcal{X}^{(t)}$ to work over micro-batches instead of windows.

4 MULTI-AGENT PARTITIONING

In real scenarios, there are multiple parallel input sources, each of which can follow a different data distribution and inject data into the system at a high rate. Passing all input streams through a single partitioner will shift the bottleneck to the partitioner itself (Figure 5a). In this section, we show how we can use Dalton in a distributed environment to coordinate multiple individual partitioners.
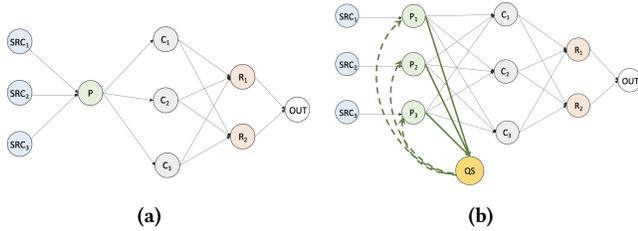

(a) **(b)**

Figure 5: (a) Dataflow topology with a single partitioner (b) Dataflow topology in the case of multi-agent partitioning. A Q-table server is used to aggregate individual Q-tables

4.1 Learning Distributed Data Streams

The high-level idea of the algorithm is that periodically, we compute and communicate to all the partitioners a global policy that is not beneficial only at an individual level but to the aggregate throughput of the system. The algorithm relies on two properties of the Q-tables: (i) they maintain information about the local heavy hitters, and (ii) according to the observed rewards, they suggest an optimal policy for the local input distribution. By averaging the individual Q-tables, we compute a global structure that incentivizes taking actions that have collected high rewards from the majority

of the partitioners. After each synchronization point, each partitioner takes actions based on this global Q-table, and eventually, the policies of different partitioners converge to a common one.

To realize the proposed algorithm, the system transparently adds a *QtableReducer (QS)* operator, one *sync* stream, shown with solid green lines in Figure 5b, and one *feedback loop* stream, shown with dashed green lines. The two added streams serve as communication channels between the *QtableReducer* and the individual partitioners. Every DSYNC time steps, each of the individual partitioners sends a SYNC message to the *QtableReducer*. This message contains: (i) the local Q-table, (ii) the total number of records processed since the last SYNC message, and (iii) a vector with the top-n most frequent keys. Once the reducer processes the SYNC messages from all the partitioners, it broadcasts back to the partitioner via the *feedback loop* channel the global Q-table, extended with an expiration timestamp of each key, and the aggregate load $\mathcal{GL}$.

Algorithm 2: Cooperative Dalton

---

local : n: number of combiners
input : (i) incoming tuple $e : (t, k, v)$ or, (ii) message from QtableReducer $(\overline{Q}, \mathcal{GL})$

1 **if** *input is* $e : (t, k, v)$ **then**
2   UpdateFrequency(k) ;
3    $f_k \leftarrow EstimateFrequency(k)$;
4   **if** $f_k \geq \frac{\mathcal{L}}{n}$ *or (k in Q and not expired)* **then**
5   assign e to $c^* = argmax_i\{Q(k, i)\}$;
6   compute reward $R(k, c^*)$;
7   **if** *state* = PREPARE **then**
8        $Q(k, c^*) = Q(k, c^*) + \gamma[R(k, c^*) - Q(k, c^*)]$;
9   **else**
10   AppendToBuffer($(k, c^*, R(k, c^*))$) ;
11   **else**
12   assign e to $c^* = hash(k)$;
13   UpdateWorkerLoad(c^*) ;
14   **if** *time from last sync* = DSYNC **then**
15   SendSyncMsg($Q, \mathcal{L}$, GetTopKeys()) ;
16   state = AWAIT ;
17 **else if** *input is* $(\overline{Q}, \mathcal{GL})$ **then**
18    $Q = \overline{Q}, \mathcal{L} = \mathcal{GL}$;
19   AggregateBufferedRewards() ;
20   state = PREPARE ;

Algorithm 2 presents the pseudocode of a Dalton operator running in a distributed setup with many partitioners. Each of the $\mathcal{P}$ partitioners can be in one of two distinct states: PREPARE and AWAIT. While in the PREPARE state, a partitioner is individually learning by taking actions and updating its local Q-table as described in Section 3.2. As soon as it emits the SYNC message, the partitioner enters the AWAIT state in which it remains until it receives the global Q-table. While in the AWAIT state, partitioners continue to receive tuples, run locally the bandit algorithm, and assign rewards to partitioning decisions. However, instead of updating the local Q-table, the rewards received during the AWAIT phase are just buffered so that they can be merged with the global Q-table once it is received. When

the global Q-table $\overline{Q}$ and the aggregate load $\mathcal{GL}$ are received, we merge the buffered rewards using Equation 5, we update the local load estimation to $\mathcal{GL}$ and transition back to the PREPARE state.

Once the *QtableReducer* has received the synchronization messages from all n partitioners, it calculates the heavy hitters that correspond to the global distribution and the global Q-table. For the heavy hitters, the reducer computes the aggregate load $\mathcal{GL} = \sum_{i=1}^{\mathcal{P}} \mathcal{L}_i$, that was processed during the PREPARE phase and considers the keys that have a frequency greater than $\frac{\mathcal{GL}}{n}$. Since the reducer receives the n most frequent keys of each partitioner and the number of heavy hitters cannot exceed n (Theorem 3.4), no heavy hitters are missed.

For the global Q-table, the *QtableReducer* calculates a weighted average over the local Q-tables. As keys are not equally frequent in the distribution of all input streams, the weights reflect the normalized frequencies as received by each partitioner. Therefore, the update formula for a key k is:

$$\overline{Q}(k, c_i) = \frac{\sum_{j=1}^{\mathcal{P}} \frac{f_j(k)}{\mathcal{GL}} Q_j(k, c_i)}{\mathcal{P}} \forall i \in [1, n]$$

where $\overline{Q}(k, c_i)$ the global/averaged Q-value for a key k and a combiner c_i. Using the frequencies as weights, the contribution of each partitioner to the value of the global Q-table for a key k is proportional to the number of rewards it has received for it.

Our proposed synchronization mechanism achieves three important properties. *First*, it does not block execution - partitioners continue to assign tuples while in AWAIT state. *Second*, by buffering rewards received in the AWAIT state, all the learned rewards are communicated to the reducer, and thus, we fully exploit acquired experiences. *Third*, we do not allow keys that are frequent only according to a local distribution but not for the global one to be split and increase the aggregation cost. A partitioner considers a key as a heavy hitter only if it exceeds the frequency threshold for the global load $\mathcal{GL}$ or if the key is included in the global Q-table.

In the multi-agent case, we map the synchronization interval DSYNC, to the STATS_WIN window used to maintain key frequency statistics to ensure that no heavy hitters are missed by the *QtableReducer*. The value of DSYNC and, hence, the frequency in which the SYNC events are emitted, affects the efficiency of the distributed mechanism. On the one hand, a short sync period favors learning but adds synchronization and communication overheads. On the other hand, rare syncs permit individual learners to deviate from the common policy. To hit a sweet spot, we propose an adaptive communication protocol that changes the sync frequency – and, hence, the STATS_WIN – at runtime.

If DSYNC time steps have passed since the last SYNC message was sent and a partitioner is still in AWAIT state, it means that the reducer cannot keep up with the synchronization rate and by the time the updated global state is received, it is already stale. In that case, by setting a field in the next SYNC message, the partitioner requests to double the DSYNC interval. When the reducer receives the SYNC messages, it first checks whether any node has requested to double DSYNC; if that is the case, it fulfills the request and broadcasts the new value along with the next global state. At the same time, the reducer always monitors the amount of time it is idle, and if this is longer than the time for processing Q-tables, it decreases DSYNC.

Discussion. We use multi-agent Dalton for optimizing a single query. However, it can be trivially extended for multiple concurrent queries over the same stream. If these queries are executed independently of each other, the only difference is that the QTableReducer should not be a query's operator but implemented in the system's coordinator. Nevertheless, when multiple aggregates are processed over the same stream, we can do much more than having a joint partitioner. Common practice suggests a global plan with shared streams and operators [19]. In such a work-sharing system, Dalton can be yet another operator of the global plan and the implementation will remain exactly as described in Sec.4.1.

4.2 Optimizing for Non-Heavy Hitters

(a) 1 partitioner: key-forwarding
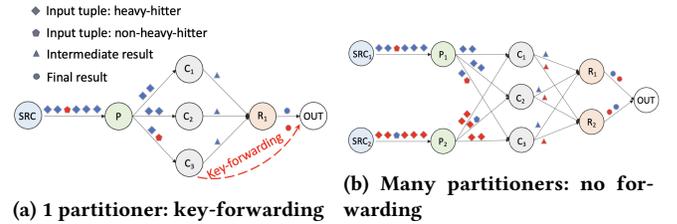
(b) Many partitioners: no forwarding

Figure 6: Execution with and without key forwarding

In the tuple-at-a-time model, when we have a single partitioner, as in Figure 6a, non-heavy hitters are hashed and are directly forwarded to the output, avoiding the extra latency from the final aggregation step. We call this scheme "*key-forwarding*". However, this cannot trivially happen in the distributed, multi-agent case. As different partitioners may observe different distributions, they cannot safely instruct a combiner to forward a key directly to the output. In Figure 6b, key k_1 (signified with the red color) is "hot" according to the global distribution. Before synchronization happens, partitioner p_1 hashes k_1 and marks it as a non-heavy hitter. Nevertheless, as p_2 has already identified k_1 as "hot" and split it, we should not forward it to the output but rather to the reducers. For this reason, in the default multi-agent implementation, we disable the "*key-forwarding*" feature. Non-heavy hitters are still hashed to avoid overloading the bandit learner but are always aggregated at the reducers for correctness. In such cases, as the burden to reducers is increased, they should be scaled out appropriately.

For the special case, where synchronization occurs at least once per slide, we propose an optimization that enables "*key-forwarding*" in the multi-agent case. Synchronizing before the window completes allows to repair wrong forwarding decisions before emitting a result. For the example of Figure 6b, the partitioner p_1 receives the global Q-table before the end of the slide, and the global Q-table marks k_1 as "hot". As this happens before the end of the slide, the window is not completed yet and combiners have not emitted their output. Thus, p_1 instructs the combiner to disable forwarding for k_1 and all intermediate results for k_1 are aggregated. A question that naturally arises is if synchronization is needed (e.g., what happens if the message from the QTableReducer is delayed). To prevent such issues, partitioners also disable forwarding if they have not received a global Q-table before the window is completed.
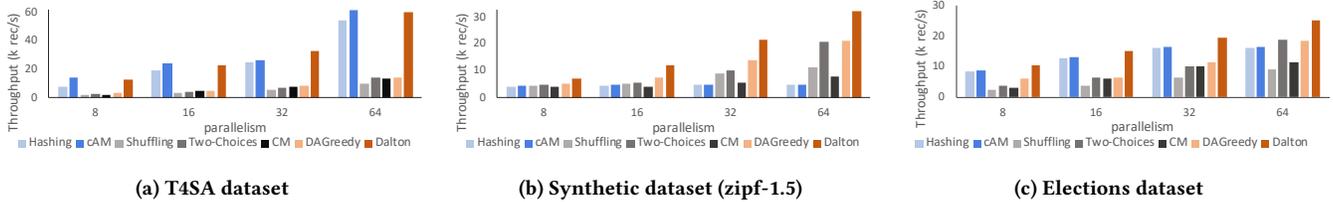
(a) T4SA dataset   (b) Synthetic dataset (zipf-1.5)   (c) Elections dataset

Figure 7: Word Count: Scalability for different distributions. window = 60*sec*, **slide**=1*sec*

(a) T4SA dataset   (b) Synthetic dataset (zipf-1.5)   (c) Elections dataset
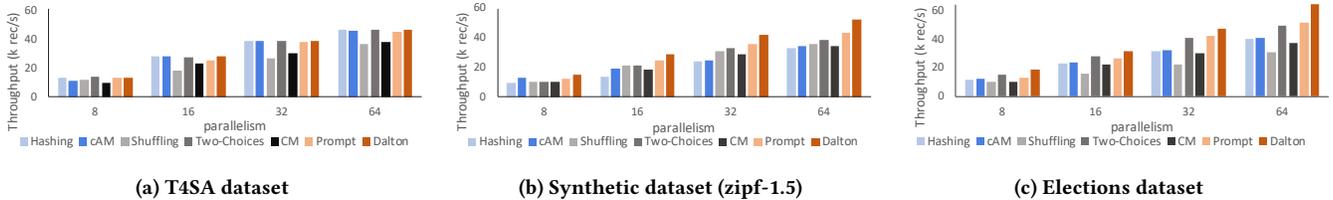
Figure 8: Micro-batch model - Word Count: Scalability for different distributions. window = 60*sec*, **slide**=1*sec*

5 EXPERIMENTAL EVALUATION

This section experimentally evaluates the scalability and adaptivity of Dalton when we vary the data distribution and the degree of parallelism. We also perform a sensitivity analysis that shows how the tuning knobs of Dalton affect performance.

Platform We use 5 two-socket Intel Xeon E5-2660 CPUs, 2.20 GHz servers with 8(×2) threads per socket and featuring 128 GB of DRAM. Tuple-based algorithms are implemented in Flink v1.12 and micro-batch in Storm Trident v2.4.0 using Java 11.0.9.1. We dedicate one server to the JobManager/Nimbus and use the remaining four for the TaskManagers/Supervisors for Flink and Storm respectively. In all the experiments, we use the tuple-based implementation unless otherwise specified.

Methodology Data is pre-loaded in main memory and is continuously consumed in a circular manner. We allow the system to warm up and measure the sustainable input throughput only after the system has been stabilized. This input rate achieves maximum utilization while ensuring that there is no backpressure.

Algorithms We compare Dalton against:

(1) *1-choice partitioners* assign all tuples with the same key to the same worker. From this group, we consider *Hashing* and *Group Affinity with Imbalance Minimization* (cAM) [21]

(2) *N-choice partitioners* apply key-splitting following a static policy for all keys. We consider *Shuffling*, *Two-choices*[3] [31], and, *Cardinality Imbalance Minimization*[4] (CM) [21].

(3) *Hybrid partitioners* split the most frequent keys and hash the rest. We consider DAGreedy [32][5] for the tuple-at-a-time processing model and Prompt [2][4] for the micro-batch model. To isolate the partitioning algorithm from the actual implementation, we implement our optimization for the non-heavy hitters (Section 4.2) for *DAGreedy* as well.

For Dalton, we set the step-size parameter $\gamma = 0.1$, the STATS_WIN interval equal to one slide and the cost model parameters $p_1 = p_2 =$

0.5 based on our experimental evidence. For the frequency statistics, we experiment with a common hashmap, a count-min sketch [11], and a hybrid policy that dynamically selects one of the two, at runtime, based on the statistics of the previous STATS_WIN interval.

Table 2: Summary of data characteristics

Dataset	# of keys	Frequency of top-1 key
T4SA	~450k	2.69%
Elections	~200k	7.2%
Voters	100k	up to 38.45%
Synthetic	100k-1M	up to 38.45%

Data To investigate the impact of different distributions, we experiment with real and synthetic data. We consider two Twitter datasets, *T4SA* [41] and *Elections [14]*, and the *voters* dataset, which represents the voter registry for North Carolina. For Twitter, we use the hashtag as the key, and for voters the post-code. Table 2 shows information for each dataset. For the synthetic data, we investigate uniform and Zipf distributions with various exponents.

Applications The majority of the experiments are based on *Word Count* as it represents a typical windowed aggregation example. As partitioning should be more lightweight than the application itself, in *Word Count*, we do not assume tuples directly in a key-value form, but parsing and key extraction are part of the application. In addition, to stress our reward model, we use *Correlation Clustering*, a common data mining task. We use the VOTE [12] algorithm for the combiners and the GREEDY algorithm [16] for the reducers. Thus, this application has quadratic complexity in the combiners and a much heavier final aggregation than the typical group by queries. The quadratic complexity acts as an adversarial example to our linear reward function. Except if mentioned otherwise, we use sliding windows with a size of 60s and a slide of 1s and 20s for *Word Count* and *Correlation Clustering*, respectively. We use the Twitter datasets for *Word Count* and *Voters* for *Correlation Clustering*.

---

[3]For Two-choices and CM, we use 2 hash functions
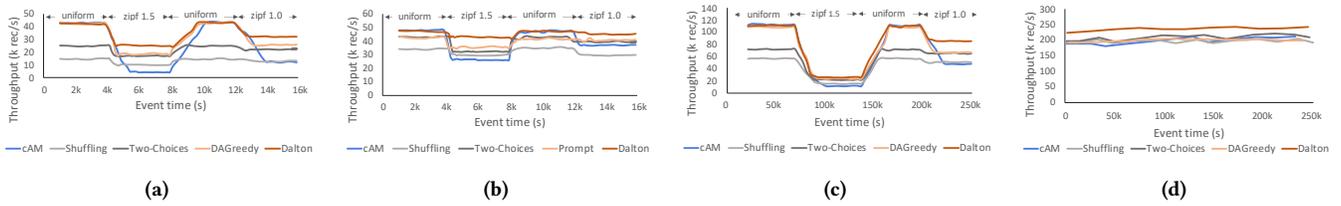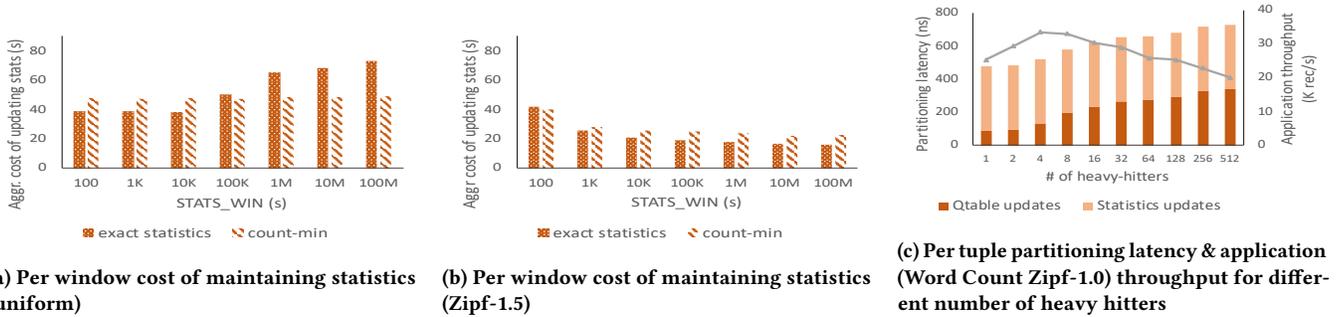[4]As there is no code available, for both systems, we used our own implementation.

Figure 9: Adaptivity to distribution shifts. (a) WordCount – Synthetic; (b) WordCount - Synthetic - Micro-batch model (c) Correlation clustering – Voters; (d) WordCount Synthetic – Zipf with a variable exponent *s*, sampled uniformly at random from [0.5, 1.5]**. Distribution changes every** 1000*s*

(a) Per window cost of maintaining statistics (uniform)

(b) Per window cost of maintaining statistics (Zipf-1.5)

(c) Per tuple partitioning latency & application (Word Count Zipf-1.0) throughput for different number of heavy hitters

Figure 10: Impact of STATS_WIN and of the number of heavy hitters to Dalton's latency

5.1 Scalability with the Number of Combiners

Figure 7 and 8 show how Word Count scales for different datasets for the tuple-at-a-time and the micro-batch processing model respectively. For the algorithms that require the final aggregation step, we use 1, 2, 4, or 8 reducers, respectively, for parallelism of 8, 16, 32, and 64, and devote the rest of the resources to combiners. The T4SA dataset is close to uniform, Elections is skewed, whereas the synthetic one is configured to present an even higher degree of skewness. We observe that hash-based algorithms scale well for uniform data but do not exploit parallelism for highly skewed workloads; adding more resources does not result in higher throughput. In contrast, algorithms that use key-splitting spread the load to combiners and solve the imbalance problem. Nevertheless, they cannot scale in the uniform case as they cause over-splitting and pay a high aggregation cost at the reducers. Existing techniques cannot scale in both uniform and skewed distributions. This is a huge problem as the partitioning algorithm is selected before launching a task and, hence, before knowing the data distribution, and also the distribution changes at runtime. In the micro-batch model the combiners compute partial aggregates per batch and not per window. Hence, even the hash-based approaches require a final aggregation step which results in a smaller difference in the performance between hash-based and key-splitting algorithms.

Takeaway. Dalton scales almost linearly regardless of the distribution. In the case of uniform data, it applies minimal splitting and behaves almost like hashing, while in the Zipf case, it discovers a policy that outperforms existing approaches by 1.5× to 6.7× for the tuple-at-a-time and 1.6× to 2.1× for the micro-batch model.

5.2 Adaptivity to Distribution Changes

Next, we showcase the ability of each algorithm to adapt to dynamic workloads. We consider two types of distribution changes: i) distribution alternates between uniform and Zipf. This scenario simulates the sporadic occurrence of trending/hot events. ii) Random changes between different Zipf distributions with different degrees of skewness and different set of heavy hitters.

The first case is illustrated in Figures 9a and 9c for the tuple-at-a-time model and the Word Count and Correlation Clustering task and in Figure 9b for the micro-batch model and Word Count task. When transitioning to a Zipf distribution, performance drops for all algorithms. However, Dalton better absorbs the change, and while the distribution is skewed, it outperforms the other algorithms by 1.3× to 6× and 1.1× to 1.8× for Word Count for the tuple-at-a-time and the micro-batch model respectively and by 1.1× to 1.8× for Correlation Clustering in the tuple-at-a-time model. Note that only Dalton and DAGreedy can adapt, while Dalton outperforms DAGreedy for skewed workloads.

For the second case, illustrated in Figure 9d, as distribution changes happen frequently and transitions are restricted among Zipf distributions, the transition points are not that visible – an averaging effect is produced. However, by learning the appropriate policy and by quickly adapting to the changes, Dalton achieves 1.1× to 1.3× higher throughput.

Takeaway. By continuously learning, Dalton adapts its policy following distribution shifts. It achieves the performance of hashing for uniform workloads and outperforms existing techniques for skewed ones in tasks with completely different computation traits.

5.3 Overhead of Partitioner

Next, we assess how Dalton's tuning knobs affect the overhead it introduces. Figures 10a and 10b show the cost of maintaining the frequency statistics as a function of the STATS_WIN parameter. The cost is the aggregate time required for updating the statistics for the processing of a window with $100M$ elements. In Figure 10a, where the distribution is uniform, STATS_WIN has no impact when the Count-Min sketch is used. On the contrary, when using a hash-map, higher STATS_WIN values translate to more keys in the map and, consequently, more cache misses that deteriorate performance. Therefore, there is no clear winner between exact computation and sketches, but the answer depends on STATS_WIN and on the number of keys. Note that while the relative difference between the two data structures is seemingly small, as it translates to latency overhead, delaying the window result by several seconds can significantly lower the quality of service. Figure 10b shows that this effect in the Zipfian case is milder due to more cache-friendly behavior (the same hot keys appear over time). To minimize the overhead, Dalton alternates between the two data structures at runtime.

Figure 10c shows: i) the latency that Dalton introduces as a function of the number of keys considered for splitting, and ii) the corresponding end-to-end application throughput. Increasing the number of heavy hitters up to 4 leads to lower load imbalance and, hence, higher throughput. For more heavy hitters, the partitioning latency is increased, affecting throughput. This justifies our decision to partition only the heavy hitters using the learner and showcases the effectiveness of our defined threshold for heavy hitters. For this experiment, we use a Zipf distribution with $s = 1.0$ to allow for more than 600 distinct keys per slide. For this distribution and according to our Definition 3.3 for heavy hitters, Dalton would consider 4 heavy hitters and, thus, achieve the maximum throughput.
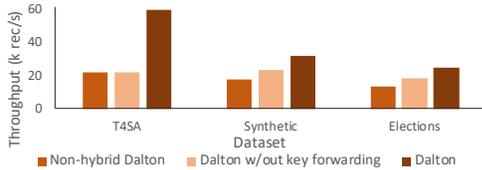

Figure 11: Contribution of individual optimizations to Dalton's performance (Word Count)

Figure 11 shows the impact of two of Dalton's optimizations on the performance of WordCount. The *Non-Hybrid* variant is our baseline; it considers all keys as heavy hitters, and hence all tuples go through the bandit. The one without key forwarding distinguishes heavy and non-heavy hitters but does not consider the forwarding of Section 4.2. Enabling heavy hitter tracking gives a 1.2× speedup on average. Finally, we enable key forwarding and have the full Dalton system. As expected, key forwarding is particularly useful in uniform distributions (T4SA) where it achieves a 3×.

Figure 12 shows the end-to-end throughput for word count at different input rates. For lower input rates, parallelism affects throughput to a lesser extent since the input throughput can be sustained with fewer workers. When the input rate is 40k rec/s, none of the algorithms can sustain the input rate and they reach their peak.

Takeaway. Choosing the right data structure for the frequency statistics can reduce the window latency by up to 24 *sec*, while using a hybrid approach that only splits the most frequent keys as well as key-forwarding results in a speedup of up to 2.7×. This is of significant importance in a streaming system with low latency requirements and high input rates.
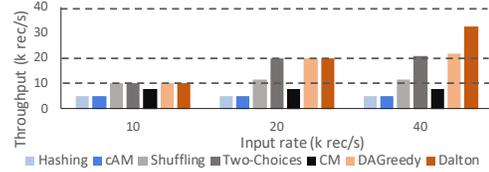

Figure 12: Throughput for varying input rates

5.4 Scaling the Partitioners

We experiment with setups with multiple input sources and partitioners. We use the task of Word Count and a window with a size of $60sec$ and a slide of $20sec$ to allow for high throughput that showcases the benefits of having multiple partitioners.
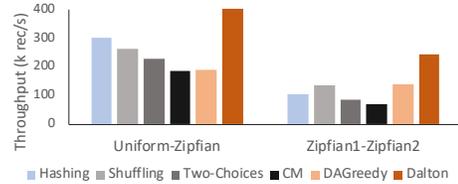

Figure 13: Word count for different distribution per source

Figure 13 shows the performance of different algorithms for two setups: i) one source is producing a uniform and the other a Zipf distribution, and ii) both sources produce a Zipf but each with different heavy hitters. We use 2 partitioners and for all Zipf distributions $s = 1.5$. As in our infrastructure, with 2 Dalton instances, partitioning is never the bottleneck, scaling further does not yield any improvement. When at least one distribution is uniform, hash-based algorithms behave better and outperform DAGreedy, while when both are Zipf, the contrary happens. In both cases, by appropriately coordinating the learners, Dalton converges to the best global policy and outperforms existing techniques by 1.4× to 3.4×.

Sync frequency. Figure 14a shows the performance of *Dalton* depending on the synchronization frequency of the partitioners DSYNC. We use two partitioners, each consuming data from a different source. We test 2 scenarios: i) one source produces data with a uniform and the other with a Zipfian distribution (s=1.5), and ii) both produce data with the same distribution. The second scenario is equivalent to producing data from a uniform and a Zipfian distribution in an alternating fashion. When the partitioners never sync, the throughput is low since they optimize only locally; being agnostic to the real load of the combiners, they give inaccurate rewards to the bandit agent. More frequent synchronization improves performance. However, the synchronization overhead increases when DSYNC is higher than 10*sec* causing performance degradation.
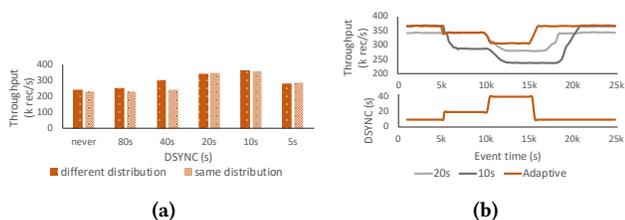
Figure 14: Synchronization frequency (DSYNC) experiments
using two partitioners. (a) Impact of DSYNC to throughput for
different distributions; (b) Dalton's policy for dynamically
updating DSYNC, under varying QTableReducer latency

Figure 14b shows how the protocol we propose for dynamically
adjusting the synchronization interval works. The top of the figure
shows throughput with the adaptive protocol and a fixed interval
of 20s and 10s. The bottom of the figure depicts the changes in
the value of DSYNC with the adaptive protocol. The initial value of
DSYNC is 20s. Initially, the partitioners observe that they receive
fast messages from the *QTableReducer* and propose more frequent
synchronization until *DSYNC* converges to 10s. This step happens
during the warmup of the system and, hence, is not depicted in the
figure. At 5000s, we artificially double the time the *QTableReducer*
needs to aggregate the *Qtables*. Hence, the partitioners propose less
frequent synchronization increasing DSYNC to 20s. At the time of
10000s, we make the time the *QTableReducer* needs for the aggrega-
tion 4 times higher than the initial, which brings DSYNC to 40s. At
15000s, we remove all imposed delays, and DSYNC returns to 10s.

Figure 15 shows the load imbalance (Equation 1) of the most im-
balanced combiner and the aggregation cost (Equation 2) imposed
by the most frequent key. In the case of multiple partitioners, half
observe a uniform distribution and half a Zipf-1.5. In the case of a
single partitioner, the input data is produced by alternating data
from a uniform and a Zipf-1.5. Learning converges to a global policy
and a stable cost in all cases. Crucially, while two partitioners have
a slightly increased imbalance compared to one, this overhead does
not increase with the number of partitioners. Moreover, not having
to pay for synchronization, the single partitioner converges faster.
However, for more than one partitioner, the rate of convergence is
not affected by the number of partitioners – as many partitioners
as necessary can be used without significantly affecting learning.

Takeaway. Our protocol successfully captures the global distri-
bution and, leveraging cooperative learning, outperforms existing
techniques by 1.4× to 3.4×. Moreover, Dalton automatically tunes
the synchronization frequency so that the communication with the
QTabeReducer does not stale execution.

6 RELATED WORK

Partitioning for streams A traditional stream partitioning ap-
proach is to dynamically re-partition in case of imbalance [35]. How-
ever, re-partitioning comes hand-in-hand with the heavyweight
task of state migration which Dalton avoids altogether. A classic
key-splitting approach is the Two-choices algorithm [31] and its
extension [30]. These algorithms address load imbalance but are
agnostic to the aggregation cost. Moreover, they make static deci-
sions and fail to adapt to distribution changes. [21] proposes a set of
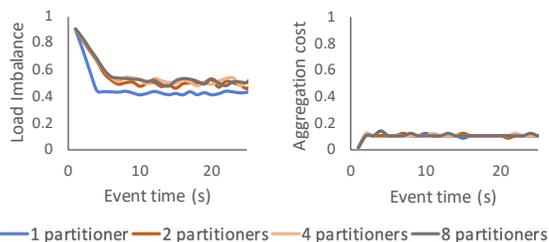

Figure 15: Convergence of Dalton for different number of
partitioners

static heuristics that consider both load imbalance and aggregation
cost. However, our evaluation shows that these heuristics do not
cover all workloads and do not always adapt to distribution shifts.

To adapt to the data, more recent approaches use dynamic strate-
gies based on the key frequencies. [13, 15] use a routing table for
heavy hitters and hashing for the rest of the keys. However, they
employ state migration, which we eliminate. DAGreedy [32] also
hashes infrequent keys and greedily assigns frequent ones based
on a cost model. Prompt [2] is a heuristic-based partitioner for the
micro-batch model. Dalton outperforms DAGreedy and Prompt by
finding better partitioning assignments and avoiding over-splitting,
and can efficiently scale. In pull-based systems, late merging can
be used instead of upfront partitioning [44]. We focus on the push-
based model, adopted by most current systems [20, 40, 43].

A related topic to partitioning is elasticity, and re-configuration
[17, 18, 34]. Such techniques can also deal with stragglers and
increase the application's throughput. Although many existing
approaches study the problem of elastically adjusting the number
of workers [2, 6], e.g., combiners and reducers, no existing work
focuses on the specific problem of scaling the partitioning operators.

Partitioning for Map-Reduce Partitioning has been widely
studied for Map-Reduce-based processing [7, 22, 23, 25]. While
conceptually similar, these approaches either require offline pre-
processing of the data and, thus, are not suitable with the streaming
model or optimize solely for the map or the reduce phase.

RL for load balancing. RL for load balancing and task sched-
uling is widely used in Cloud Computing [4, 5, 27, 39]. However,
these applications do not consider balancing among operators with
a windowed state. PartLy [1] uses deep RL for partitioning in the
micro-batch setup. However, it assumes prior knowledge of a fixed
distribution, which violates stream processing requirements.

7 CONCLUSION

The performance of stream processing systems highly depends on
the efficiency of partitioning the load among parallel workers. Re-
source underutilization and key oversplitting introduce overheads
degrading throughput. Moreover, as streams are unpredictable and
distributed in nature, to ensure high, effective parallelism, systems
should quickly adapt to the distribution at hand and be able to scale
not only the processing workers but also the partitioners. This work
presents Dalton, an RL-based operator that learns, with minimal
overhead, partitioning policies at runtime, meets both desiderata
and outperforms state-of-the-art approaches by up to 6.7×.

REFERENCES

[1] Ahmed S. Abdelhamid and Walid G. Aref. 2020. PartLy: Learning Data Partitioning for Distributed Data Stream Processing. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 6, 4 pages. https://doi.org/10.1145/3401071.3401660

[2] Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref. 2020. Prompt: Dynamic Data-Partitioning for Distributed Micro-Batch Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2455–2469. https://doi.org/10.1145/3318464.3389713

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Ali Asghari, Mohammad Karim Sohrabi, and Farzin Yaghmaee. 2020. Online Scheduling of Dependent Tasks of Cloud's Workflows to Enhance Resource Utilization and Reduce the Makespan Using Multiple Reinforcement Learning-Based Agents. *Soft Comput.* 24, 21 (nov 2020), 16177–16199. https://doi.org/10.1007/s00500-020-04931-7

[5] Ali Asghari, Mohammad Karim Sohrabi, and Farzin Yaghmaee. 2021. Task Scheduling, Resource Provisioning, and Load Balancing on Scientific Workflows Using Parallel SARSA Reinforcement Learning Agents and Genetic Algorithm. *J. Supercomput.* 77, 3 (mar 2021), 2800–2828. https://doi.org/10.1007/s11227-020-03364-1

[6] Marcos Assuncao, Alexandre Veith, and Rajkumar Buyya. 2017. Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. *Journal of Network and Computer Applications* 103 (12 2017). https://doi.org/10.1016/j.jnca.2017.12.001

[7] Joanna Berlinska and Maciej Drozdowski. 2018. Comparing load-balancing algorithms for MapReduce under Zipfian data skews. *Parallel Comput.* 72 (2018), 14–28. https://doi.org/10.1016/j.parco.2017.12.003

[8] Djallel Bouneffouf and Irina Rish. 2019. A Survey on Practical Applications of Multi-Armed and Contextual Bandits. https://doi.org/10.48550/ARXIV.1904.10040

[9] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (2014), 401–412. https://doi.org/10.14778/2735496.2735503

[10] Aaron Clauset, Cosma Shalizi, and Mark Newman. 2007. Power-Law Distributions in Empirical Data. *SIAM Rev.* 51 (06 2007). https://doi.org/10.1137/070710111

[11] Graham Cormode and Senthilmurugan Muthukrishnan. 2004. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. 29–38.

[12] Micha Elsner and Warren Schudy. 2009. Bounding and Comparing Methods for Correlation Clustering beyond ILP. In *Proceedings of the Workshop on Integer Linear Programming for Natural Language Processing* (Boulder, Colorado) *(ILP '09)*. Association for Computational Linguistics, USA, 19–27.

[13] Junhua Fang, Rong Zhang, Tom Z.J. Fu, Zhenjie Zhang, Aoying Zhou, and Junhua Zhu. 2017. Parallel Stream Processing Against Workload Skewness and Variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA) *(HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3078597.3078613

[14] Alex Filatov. 2020. 2016 USA Presidential election tweets. https://data.world/alexfilatov/2016-usa-presidential-election-tweets. Accessed: 2022-06-01.

[15] Buğra Gedik. 2014. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal* 23, 4 (aug 2014), 517–539. https://doi.org/10.1007/s00778-013-0335-9

[16] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental Record Linkage. *Proc. VLDB Endow.* 7, 9 (may 2014), 697–708. https://doi.org/10.14778/2732939.2732943

[17] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman (Eds.). ACM, 276–287. https://doi.org/10.1145/2806777.2806847

[18] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. 2015. An adaptive replication scheme for elastic data stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 150–161. https://doi.org/10.1145/2675743.2771831

[19] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data*. 607–622.

[20] Asterios Katsifodimos and Sebastian Schelter. 2016. Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 193–193. https://doi.org/10.1109/IC2EW.2016.56

[21] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. 2017. A Holistic View of Stream Partitioning Costs. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1286–1297. https://doi.org/10.14778/3137628.3137639

[22] Mohamed Khafagy, Ahmed Wahdan, and Hesham Hefny. 2014. Comparative Study Load Balance Algorithms for Map Reduce Environment. *International Journal of Applied Information Systems* 7 (11 2014), 41–50. https://doi.org/10.5120/ijais14-451261

[23] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load Balancing for MapReduce-based Entity Resolution. In *2012 IEEE 28th International Conference on Data Engineering*. 618–629. https://doi.org/10.1109/ICDE.2012.22

[24] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 555–569. https://doi.org/10.1145/2882903.2882906

[25] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2213836.2213840

[26] Tyler Lu, Dávid Pál, and Martin Pal. 2010. Contextual Multi-Armed Bandits. *Journal of Machine Learning Research - Proceedings Track* 9, 485–492.

[27] Fatma M. Talaat, Mohamed Sabry, Ahmed Saleh, Hesham Ali, and Shereen Ali. 2020. A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment. *Journal of Ambient Intelligence and Humanized Computing* 11 (11 2020). https://doi.org/10.1007/s12652-020-01768-8

[28] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[29] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[30] Anis Nasir, Gianmarco Morales, Nicolas Kourtellis, and Marco Serafini. 2016. When two choices are not enough: Balancing at scale in Distributed Stream Processing. 589–600. https://doi.org/10.1109/ICDE.2016.7498273

[31] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*. 137–148. https://doi.org/10.1109/ICDE.2015.7113279

[32] Anil Pacaci and M. Tamer Özsu. 2018. Distribution-Aware Stream Partitioning for Distributed Stream Processing Systems. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Foto N. Afrati, Jacek Sroka, Ke Yi, and Jan Hidders (Eds.). ACM, 6:1–6:10. https://doi.org/10.1145/3206333.3206338

[33] Nicolo Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 80–91. https://doi.org/10.1145/2675743.2771827

[34] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Comput. Surv.* 52, 2 (2019), 36:1–36:37. https://doi.org/10.1145/3303849

[35] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. 2003. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). IEEE Computer Society, 25–36. https://doi.org/10.1109/ICDE.2003.1260779

[36] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[37] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2019. Sliding-Window Aggregation Algorithms. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. https://doi.org/10.1007/978-3-319-63962-8_157-1

[38] Khin Me Me Thein. 2014. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research* 3, 47 (2014), 9478–9483.

[39] Zhao Tong, Xiaomei Deng, Hongjian Chen, Jing Mei, and Hong Liu. 2020. QL-HEFT: A Novel Machine Learning Scheduling Scheme Base on Cloud Computing Environment. *Neural Comput. Appl.* 32, 10 (may 2020), 5553–5570. https://doi.org/10.1007/s00521-019-04118-8

[40] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 147–156. https://doi.org/10.1145/2588555.2595641

[41] Lucia Vadicamo, Fabio Carrara, Andrea Cimino, Stefano Cresci, Felice Dell'Orletta, Fabrizio Falchi, and Maurizio Tesconi. 2017. Cross-Media Learning for Image Sentiment Analysis in the Wild. In *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*. 308–317. https://doi.org/10.1109/ICCVW.2017.45

[42] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (1992), 279–292. https://doi.org/10.1007/BF00992698

[43] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

[44] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (jan 2019), 516–530. https://doi.org/10.14778/3303753.3303758