PartLy: Learning Data Partitioning for Distributed Data Stream Processing

Ahmed S. Abdelhamid and Walid G. Aref
Department of Computer Science, Purdue University
West Lafayette, Indiana
{samy,aref}@purdue.edu

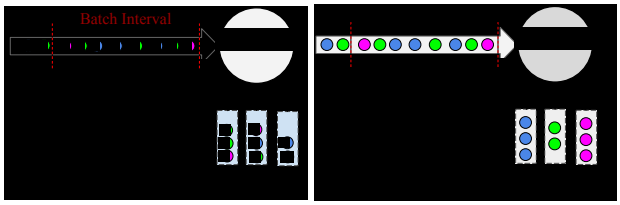
ABSTRACT

Data partitioning plays a critical role in data stream processing. Current data partitioning techniques use simple, static heuristics that do not incorporate feedback about the quality of the partitioning decision (i.e., fire and forget strategy). Hence, the data partitioner often repeatedly chooses the same decision. In this paper, we argue that reinforcement learning techniques can be applied to address this problem. The use of artificial neural networks can facilitate learning of efficient partitioning policies. We identify the challenges that emerge when applying machine learning techniques to the data partitioning problem for distributed data stream processing. Furthermore, we introduce PartLy, a proof-of-concept data partitioner, and present preliminary results that indicate PartLy's potential to match the performance of state-of-the-art techniques in terms of partitioning quality, while minimizing storage and processing overheads.

ACM Reference Format:
Ahmed S. Abdelhamid and Walid G. Aref. 2020. PartLy: Learning Data Partitioning for Distributed Data Stream Processing. In *Third Workshop in Exploiting AI Techniques for Data Management (aiDM'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3401071.3401660

1 INTRODUCTION

Data partitioning is a well-studied problem in distributed stream data processing [5–7, 12, 13]. The basic partitioning techniques are *shuffling*, and *hashing*. In shuffle partitioning, data tuples are assigned to processing nodes in a round-robin fashion based on the order of arrival (see Figure 1a). Shuffle partitioning guarantees that all the processing nodes receive an almost equal number of data tuples. However, it does not insure *key locality*, i.e., tuples with the same key are not necessarily sent to the same processing node, and thus, increasing the overhead in transferring data to compute per-key aggregates (see [13] for details). In contrast, hash partitioning, also termed *Key Grouping* [13], applies a hash function over one or more particular fields of each tuple, i.e., a *partitioning key*, to route the tuple into a processing node (see Figure 1b). Thus, hash partitioning assigns all the data tuples with the same keys to the same processing node. However, in case the input data stream is skewed, some key values will appear more often than others. Thus, hash partitioning would result in unbalanced input to the processing nodes. Both techniques lead to resource under-utilization and performance degradation. The state-of-the-art in stream data partitioning techniques applies static heuristics to achieve the benefits of both the shuffling and the hashing techniques, while minimizing their drawbacks. One example is to split the skewed keys only over multiple nodes [12, 13]. In order to achieve that, the data partitioner applies multiple hash functions to the tuple's *partitioning key* to generate multiple candidate assignments for the data tuple. Then, the partitioner selects the node with the least number of tuples at the time of the decision. In order to realize this objective, the partitioner maintains the following two statistics in real-time: (1) The number of tuples assigned to each processing node, and (2) Counts on the input data distribution to detect the skewed keys and split them. These partitioning techniques rely on static heuristics and do not learn from previous experiences. The data partitioner never learns from previous good or bad decisions.



(a) Shuffle Partitioning **(b) Hash Partitioning**

Figure 1: Data Partitioning Techniques

aiDM'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8029-4/20/06…$15.00
https://doi.org/10.1145/3401071.3401660

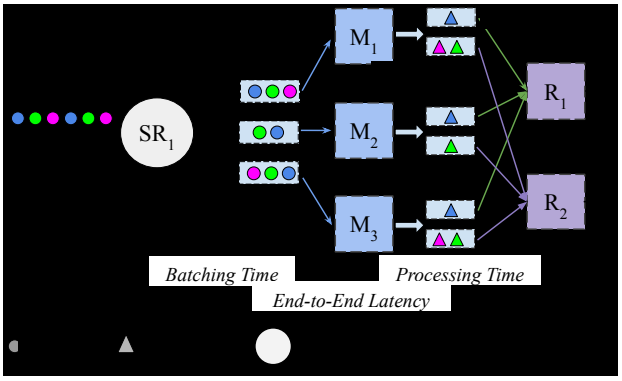


Figure 2: Example of micro-batch stream processing with 3 Map and 2 Reduce tasks, and a Stream Receiver (SR_1) with PartLy to partition a micro-batch into data blocks.

In this paper, we present our vision of a learning-based data partitioner that leverages prior experience, aiming to learn how to partition future data tuples more effectively (i.e., for better load-balancing) and efficiently (e.g., without the counting structures). We apply reinforcement learning that has been successfully used in various data management problems including query optimization, indexing, and query scheduling (e.g., [8, 10, 11]). Deep reinforcement learning is a process by which an agent learns a task through continuous feedback with the help of a neural network. Existing machine learning techniques can provide effective load-balanced data partitioning with less counting overhead. To the best of our knowledge, this work is the first to realize a data stream partitioner using reinforcement learning. Section 2 presents the challenges in adopting learned data partitioning for data stream processing. Section 3 introduces PartLy, a learned data partitioner that relies on deep reinforcement learning [2]. Section 4 presents preliminary results that demonstrate PartLy's potential to match state-of-the-art techniques. Section 5 describes our ongoing and related work.

2 CHALLENGES

In this section, we identify several important challenges when applying learning techniques to the data partitioning problem.

2.1 Real-time Processing

The real-time execution in data stream processing systems requires the partitioning techniques to make a swift per-tuple decision upon tuple arrival. Otherwise, data partitioning may lead to performance bottlenecks by increasing end-to-end tuple processing times. In addition, the input data rate is typically in the order of millions of tuples per second (e.g., see [14]). Processing individual tuples through a neural network in real-time is challenging. One possible solution is to use micro-batched stream processing (e.g., [15]) to amortize the cost over a group of tuples. In contrast to tuple-at-a-time

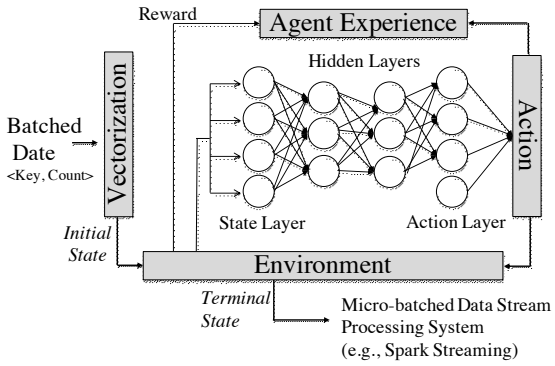


Figure 3: Proposed PartLy Design

stream processing, the partitioning decision is taken collectively for a group of tuples that are buffered within a batch. Hence, the data tuples are assigned to data blocks and consequently each data block is assigned to a processing node. Furthermore, PartLy operates on key value of tuples within a batch (i.e., one decision is given to all tuples sharing the same key value within a micro-batch).

2.2 Decision Space

The number of possible assignments of data tuples within a batch to processing nodes is exponential (i.e., M^K where M is the number of processing nodes, and K is the number of distinct keys). In this version of PartLy, we restrict the partitioning of one key value to only two processing nodes (as in [13]). This reduces the decision space to $2 * K * M$. PartLy adopts [13]'s cost model that uses the number of tuples assigned to a processing node to calculate the reward for the training episodes.

2.3 Random Data Arrival

The data partitioner should process randomly-arriving data tuples over time. Training reinforcement learning algorithms requires "training" episodes with finite time horizon. The randomness in streamed data distribution creates difficulty in training due to the variance in computing the reward of episodes. Due to this randomness, each micro-batch often contains a different number of keys with different counts. PartLy uses a recent technique for dynamic environments [9], where the running-average is used to compute the reward over the episodes.

3 DESIGN OF PartLy

3.1 The Data Partitioning Problem

The distributed micro-batched stream processing model executes a continuous query using consecutive, independent, and stateless Map-Reduce tasks over small batches of streamed data. Figure 2 shows the physical details of execution, e.g., the level of parallelism (the number of Map and Reduce tasks, data partitioning), the order of task execution, and data dependencies among the tasks. Each micro-batch is partitioned

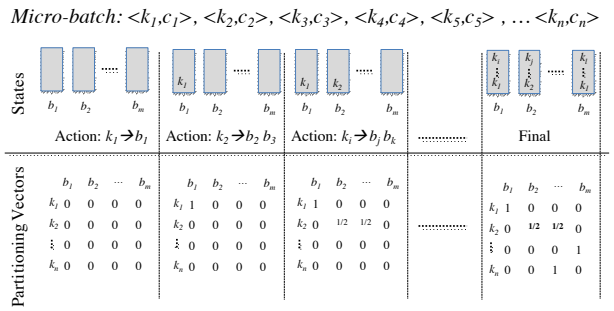


Figure 4: Action in Training Episodes

based on the supported level of parallelism. We term every partition a *data block*. Let b_i be the ith data block. The input data is an infinite stream of tuples. Each tuple has a key k that is used to guide partitioning. The data partitioning problem can be defined as the strategy to assign data tuples to data blocks according to some criteria. For each micro-batch, PartLy aims to provide even input to all processing nodes to maximize throughput and system utilization. PartLy's data partitioning scheme has the following constraints: (1) The *batch size* is a system parameter to meet an end-to-end latency required by the user. (2) Computing resources are fixed, i.e., the number of processing nodes is a user parameter.

3.2 State Representation

PartLy uses a vector for each state to represent information about the batched data and the assignment to data blocks. Each state represents a partial assignment of keys to data blocks. Each vector is a row of size n, where n is the number of keys in the batch. Let c_i be the number of tuples having Key k_i in the batch. c_i is set to 0 when a key is fully assigned to one or two data blocks. The assignment of keys to data blocks is captured using a matrix M of size $n * m$ for each episode. The value M_{ij} is 1 if k_i is assigned to $Block_j$. This value is 0 if no tuples of k_i is assigned to $Block_j$. $M_{ij} = 0.5$ if k_i is split across two data blocks. Figure 4 shows a generic representation of an episode in PartLy.

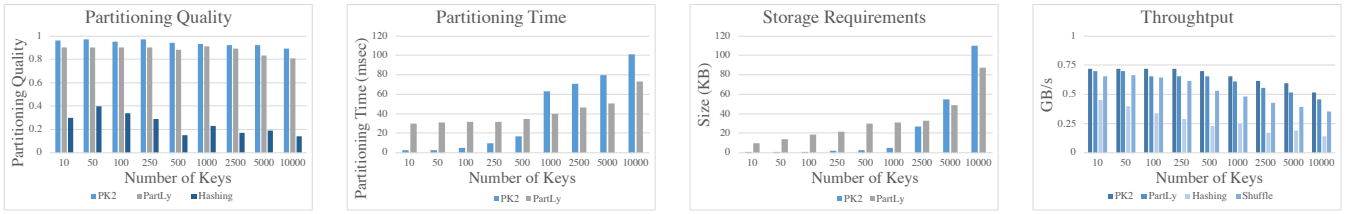
3.3 Training Process

PartLy uses deep reinforcement learning, where an agent interacts with the defined environment (See Figure 3). The environment informs the agent of its current state, s_t, and the set of potential actions $A_t = \{a_0, a_1, ..., a_n\}$ that the agent can choose from. The agent executes an action $a \in A_t$, and the environment responds to the agent with a reward r_t. The environment provides the agent with a new state s_{t+1} and a new action set A_{t+1} that reflects the status after the recent action. This process repeats until a terminal state is reached (i.e., when no more actions are available to execute). This marks the end of an episode after which a new episode may begin. The objective of an agent is to maximize the reward over episodes by learning from the agent's previous actions.

PartLy treats every batch of data as an episode, and learns continuously over the multiple batches. PartLy uses a policy gradient method to select actions based on Policy π_θ (i.e., neural network), where θ is a vector of policy parameters. The policy π_θ is optimized over episodes by modifying its parameters θ (i.e., the neurons' weights) to generate the best reward. PartLy uses the cost model of the partitioner in [13] to compute the rewards of episodes. The cost model relies on checking the difference in sizes between the maximum and average data blocks. The agent's objective is to minimize this difference through maximizing the reward. Hence, the reward is set to the negative of the difference between the maximum size and the average size of all generated data blocks for the batch: $max|Block_i| - avg|Block_i| \quad i \in p$, where p is the number of data blocks. Notice that, while the agent is trying to generate even-sized data blocks, the action space only allows a key to be assigned to one data block or split over two data blocks, and thus preserving key-locality to two data blocks in the worst case. Figure 3 gives an overall view of PartLy. The micro-batch statistics (i.e., the list of key counts) is vectorized, and is inserted into the state layer. The statistics are collected using an online technique while buffering the data tuples similar to the one discussed in [1]. Count values of the distinct keys are transformed and are passed to hidden layers, and finally to the action layer. In each experimental setup, PartLy assumes a maximum number of keys. If the number of keys within a batch is less than expected, their counts are set to 0. The output of the action layer is normalized to form a probability distribution to allow for action selection. Rewards are computed only for a terminal state, i.e., when all keys are assigned. The intermediate states have a zero reward. In addition, the final reward is computed using a running average over the previous episodes to reduce randomness effect and promote generating a general policy. To train the model, PartLy uses the *Proximal Policy Optimization (PPO)* algorithm [3] within TensorForce [4]. Training takes around 20,000 simulated one-second batches of data, and this is analogous to 5.5 hours in an actual setup).

4 PRELIMINARY EVALUATION

We present some promising results that demonstrate PartLy's ability to generate sound partitioning. In the experiments, we use the *WordCount* query that performs a sliding window count over 30 seconds over a stream of tweets. For data partitioning, each tweet is split into words that are used as the keys for the tuple. The query is written in map-reduce. Experiments are conducted for an execution setup of 5 nodes with 8 cores each (i.e., the number of data blocks is 24). Apache Spark v2.0.0 is the processing engine. We generate batches with different number of keys (i.e., ranging from few keys to thousands of keys). We assess the effectiveness of PartLy against traditional and



(a) Partitioning Quality **(b) Partitioning Time** **(c) Storage Requirements** **(d) Throughput**

Figure 5: PartLy Partitioning Effectiveness

state-of-the-art techniques: *Shuffle*, *Hashing*, and *PK-2* [13]. Figure 5a compares the *partitioning quality* metric for all the techniques. The partitioning quality is defined as $max|Block_i| - avg|Block_i| \quad i \in p$, and is computed relative to the Shuffle technique, where size balancing is guaranteed at the expense of broadcasting keys to all data blocks (i.e., potentially increased overhead at the compute nodes). PartLy demonstrates the ability to match PK2's performance [13] in providing load-balanced partitions. We verify the partitioning strategies for both algorithms by feeding the generated data blocks to the Spark Streaming engine. The difference in latency between PartLy and PK2 is below 5% for this workload. Figure 5d demonstrates PartLy's ability to maintain a competitive throughput while changing data distribution. Figure 5b gives the partitioning cost in terms of the required time to partition a micro-batch into data blocks. PartLy shows potential to outperform PK2 [13] w.r.t. speed as the number of keys increases. Thus, PartLy's ability to provide a sound partitioning strategy in less time is promising. Also, from Figure 5c, PartLy requires less space in contrast to PK2's increased demand for book-keeping as the number of keys increases.

5 FUTURE DIRECTIONS

PartLy demonstrates that there is potential for applying reinforcement learning to the data partitioning problem, which opens exciting research directions as we highlight below:

Run-time Optimization: We plan to use the actual latency of executing a computation on Spark Streaming to compute the reward for the training algorithm. PartLy uses PK2's cost model [13] to bootstrap the training process for a large number of episodes. We plan to enrich the learning process by mimicking other techniques with richer action spaces, e.g., ones where cardinality and aggregation costs of data partitioning decision are considered (e.g., [1, 5]). In addition, we plan to explore better representations for the action space, e.g., to allow the model to split a key over a larger number of processing nodes or with varying ratios. Moreover, we plan to study the learning ability of PartLy under various queries and dynamic workloads.

Learned Elastic Scheduling: We plan to expand PartLy to allow for a dynamic number of data blocks, hence enabling learned elasticity. For instance, PartLy can utilize the relationship between the batching time and the processing time of previous batches to guide elasticity for the incoming batches. This would allow PartLy to adjust the number of data blocks to meet the user's requirements (e.g., to enforce a target latency as part of a Service Level Agreement).

ACKNOWLEDGMENTS

Walid G. Aref acknowledges the support of the U.S. NSF under Grant Numbers: IIS-1910216 and III-1815796.

REFERENCES

[1] Prompt: Online data-partitioning for distributed micro-batch streaming systems. In *Sigmod*, 2020.
[2] A. K. et. al. Brief survey of drl. In *IEEE Signal Processing*, 2017.
[3] S. J. et al. Proximal policy optimization algorithms. In *arXiv*, 17.
[4] S. M. et. al. Tensorforce: A tensorflow library for applied reinforcement learning. In *https://github.com/reinforceio/tensorforce*.
[5] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. In *VLDB*, 2017.
[6] A. A. B. Lima, M. Mattoso, and P. Valduriez. Adaptive virtual partitioning for olap query processing in a database cluster. In *Journal of Information and Data Management*, volume 1, pages 75–87, 2010.
[7] M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez. Dynamic workload-based partitioning for large-scale databases. In *DEXA*, pages 183–190, 2012.
[8] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.
[9] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. Variance reduction for reinforcement learn- ing in input-driven environments. In *ICLR*, 2019.
[10] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. In *arXiv*, 2018.
[11] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *aiDM*, 2018.
[12] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *ICDE*, 2016.
[13] M. A. U. Nasir, G. D. F. Morales, D. G. Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, 2015.
[14] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.
[15] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.