PartLy: Learning Data Partitioning for Distributed Data Stream Processing

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

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 *aiDM '20: Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, June 19, 2020, Portland, OR.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

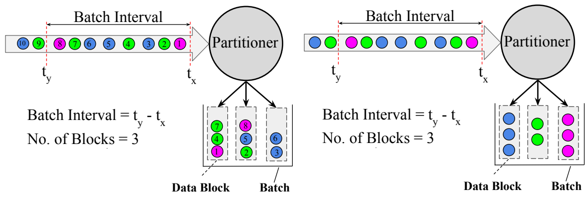
1 INTRODUCTION

Data partitioning is a well-studied problem in distributed stream data processing [2, 6, 8–11, 15, 16, 19]. 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 1aa). Shuffle partitioning guarantees that all the processing nodes receive even workload even with dynamic input data rates. However, shuffle partitioning has a major drawback. It does not insure *key locality*, i.e., tuples with the same key are not necessarily sent to the same node. In contrast, hash partitioning, also termed *Key Grouping* [16], 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. 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. One example is to split the skewed keys only over multiple nodes [15, 16]. 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.

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 use reinforcement learning that has been successfully used in various data management problems including query optimization, indexing, and query scheduling (e.g., [7, 13, 14]. 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

aiDM '20, June 19, 2020, Portland, Oregon, USA

ACM ISBN 978-1-4503-9999-9/20/06…$15.00
https://doi.org/10.1145/1122445.1122456



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

Figure 1: Data Partitioning Techniques

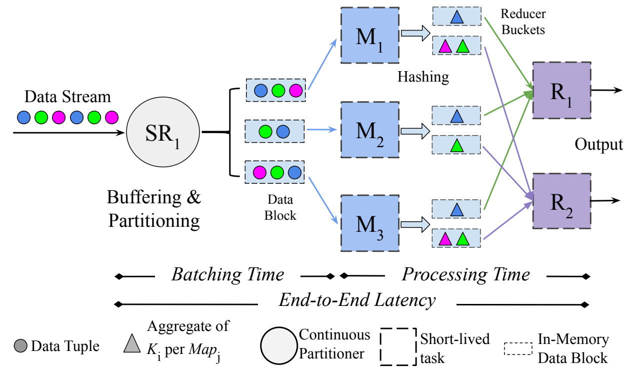


Figure 2: Example of micro-batch stream processing with 3 Map and 2 Reduce tasks, and a Stream Receiver (SR_1) with PartLy to partition a micro-batch into data blocks.

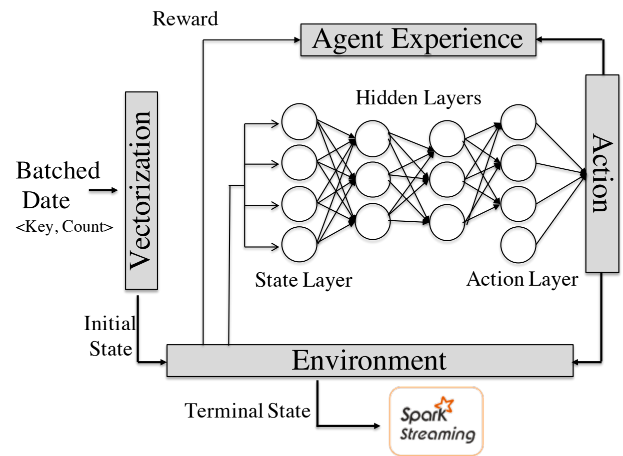


Figure 3: Proposed PartLy Design

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 [3]. 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 the challenges faced when applying learning techniques to the data partitioning problem.

2.1 Real-time Processing

The nature of execution in data stream processing systems requires the partitioning techniques to make a swift per-tuple decision upon tuple arrival. Otherwise, stream processing will be interrupted. The input data rate can be in millions of tuples per second. Processing individual tuples through a neural network in real-time is challenging. One possible solution is to use micro-batched stream processing (e.g., [18]) to amortize the cost over a group of tuples. In contrast to tuple-at-a-time 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 keys sharing the same key value within a micro-batch). This significantly decreases the overhead of PartLy.

2.2 Huge Decision Space

The number of possible assignments of data tuples within a batch to processing nodes is exponential. In this version of PartLy, we restrict the partitioning of one key value to only two processing nodes (as in [16]). PartLy adopts [16]'s cost model that uses the number of tuples assigned to a processing node to calculate the reward for the training episodes [16].

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 input-driven environments [12], 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. Each micro-batch is partitioned based on the supported level of parallelism, i.e., the number of processing nodes. We term every partition a *data block*. Let b_i be the ith data block. The input data S is an infinite stream of tuples. Each tuple $t = (t_s, k, v)$ where t_s is a timestamp set by the stream's originating source, k is a key that is used to partition the tuples for distributed processing, and v is a value that can be single or multiple data fields within the data tuple. In Figure 2, the execution graph shows the physical details of execution,

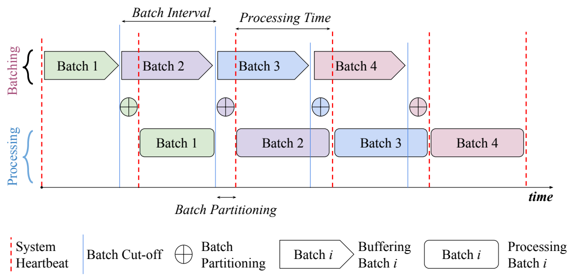


Figure 4: Processing Model

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. For each micro-batch, PartLy aims to provide even input to all processing nodes to maximize throughput and system utilization (See [17]). 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. Figure 4 gives an overview of PartLy's micro-batched stream processing model. In the batching phase, PartLy collects key counts to partition the data blocks for execution over the processing nodes.

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 v_i be the number of tuples having Key k_i in the batch. v_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 one if k_i is assigned to $Block_j$. This value is zero 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 5 shows a generic representation of an episode in PartLy.

3.3 Training Process

PartLy uses 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.

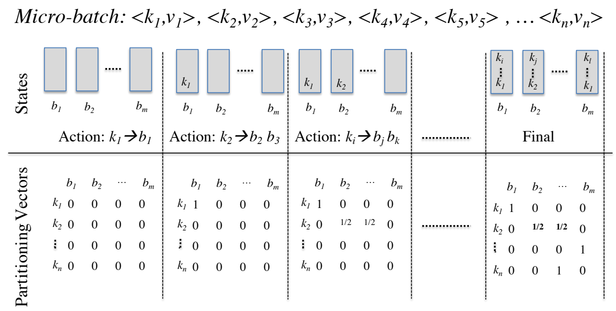
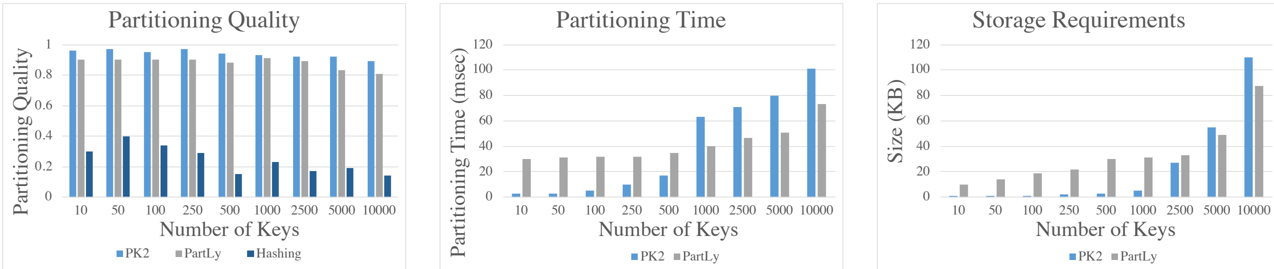


Figure 5: Action in Training Episodes

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 [16] to compute the rewards of episodes. The cost model relies on checking the difference in sizes between the largest and smallest data blocks. The agent's objective is to minimize this difference. Figure 3 gives an overall view of PartLy. The micro-batch statistics (i.e., the list of <k,v> pairs) is vectorized and is inserted into the state layer. Values are transformed and are passed to hidden layers, and finally to the action layer. 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 avoid randomness effect and promote generating a general policy. To train the model, PartLy uses the *Proximal Policy Optimization (PPO)* algorithm [4] within TensorForce [5]. Training takes around 20,000 one-second batches of data (5.5 hours).

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 40). Apache Spark v2.0.0 is the processing engine. The micro-batch interval is fixed to 1 second. We use the tweets dataset to generate batches with different number of keys (i.e., ranging from few keys to thousands of keys). We assess the effectiveness of PartLy using two metrics: *Partitioning Time* and *Partitioning Quality*. We



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

Figure 6: PartLy Partitioning Effectiveness

compare with traditional and state-of-the-art techniques: *Shuffle*, *Hashing*, and *PK-2* [16]. Figure 6a compares the *partitioning quality* metric achieved for all the techniques relative to the Shuffle technique, where size balancing is guaranteed at the expense of broadcasting keys to all data blocks (i.e., increased overhead at the processing nodes). PartLy demonstrates the ability to match PK2's performance [16] in providing load-balanced partitions. We verify the partitioning strategies for both algorithms on Spark Streaming engine. The difference in latency between PartLy and PK2 is below 5% for this workload. Figure 6b gives the partitioning cost in terms of the required time to process a micro-batch and provide a partitioning strategy. PartLy shows potential to outperform PK2 [16] 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 6c, 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 [16] to bootstrap the training process for a large number of episodes. We plan to enrich the learning process by mimicking more optimized techniques with richer action spaces, e.g., ones where cardinality and aggregation costs of data partitioning decision are considered (e.g., [1, 8, 17]). 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. Also, we plan to integrate PartLy into the blocking module of Spark Streaming to offer real-time learned partitioning.

Learned Elastic Scheduling: PartLy assumes a fixed number of processing nodes. We plan to expand PartLy to allow for a dynamic number of data blocks, hence enabling learned elasticity. This would allow the learned data partitioner to decide on the number of data blocks to match 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] C. Balkesen and N. Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *8th International Workshop on Data Management for Sensor Networks (DMSN)*, 2011.
[3] A. K. et. al. Brief survey of drl. In *IEEE Signal Processing*, 2017.
[4] S. J. et al. Proximal policy optimization algorithms. In *arXiv*, 17.
[5] S. M. et. al. Tensorforce: A tensorflow library for applied reinforcement learning. In *https://github.com/reinforceio/tensorforce*.
[6] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *International Conference on Distributed Computing Systems*, 2010.
[7] S. B. V. Z. M. M. A. Hongzi Mao, Malte Schwarzkopf. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.
[8] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. In *VLDB*, 2017.
[9] 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.
[10] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez. Data partitioning for minimizing transferred data in mapreduce. In *Globe*, 2013.
[11] 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.
[12] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. Variance reduction for reinforcement learn- ing in input-driven environments. In *ICLR*, 2019.
[13] 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.
[14] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *aiDM*, 2018.
[15] 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.

[16] 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.

[17] 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.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[19] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. In *VLDB*, 2011.