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:

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 1a). 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...
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 $i$th 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,
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 \( \pi_0 \) (i.e., neural network), where \( \theta \) is a vector of policy parameters. The policy \( \pi_0 \) is optimized over episodes by modifying its parameters \( \theta \) (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).
Partitioning Time

<table>
<thead>
<tr>
<th>Size (KB)</th>
<th>Partitioning Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.2</td>
</tr>
<tr>
<td>120</td>
<td>0.4</td>
</tr>
<tr>
<td>200</td>
<td>0.6</td>
</tr>
</tbody>
</table>

External metrics, such as the 

Figure 6 gives the partitioning effectiveness of PartLy in terms of cost, quality, and storage requirements.

(a) Partitioning Quality

(b) Partitioning Time

(c) Storage Requirements

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 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 IIS-1815796.

REFERENCES


