ML-based Dynamic Operator-Level Query Mapping for Stream Processing Systems in Heterogeneous Computing Environments

Sejeong Oh, Gordon Euhyun Moon and Sungyong Park*
Sogang University, Seoul, Republic of Korea
{sjoh2, ehmoon, parksy}@sogang.ac.kr

***Abstract*—Mapping queries to optimal computing devices at the operator-level presents a significant challenge in stream processing systems (SPS) with heterogeneous computing resources. Inefficient query mapping can degrade the performance of the SPS. To address this issue, existing approaches employ static methods, such as mapping all queries to either CPUs or GPUs, or maintaining static mapping tables for queries or operators based on their predetermined device preferences. However, the static mapping scheme fails to provide an optimal solution, as the device preference for different query operators changes dynamically at runtime. In this paper, we propose DYNO, a high performance SPS that dynamically maps queries to devices at the operator-level using a tree-based machine learning algorithm. To effectively determine an optimized device mapping plan for query operators, DYNO employs a tree-based gradient boosting model to accurately predict the execution time for all potential mapping plan combinations. DYNO also introduces a novel turn-based updating scheme to maximize performance in stream processing while training a tree-based gradient boosting model. Additionally, we devise an efficient device mapping scheme to expedite the process of determining the optimal device mapping plan by leveraging a direct acyclic graph (DAG) shortest path algorithm. DYNO completely hides any overhead caused by the extra computation needed to find the optimal plan by utilizing prefetching and GPU idle periods. Experimental results using a variety of queries and traffic patterns show that DYNO outperforms existing state-of-the-art approaches by ensuring high throughput, low latency, and high efficiency.**

***Index Terms*—heterogeneous environments, device mapping, machine learning, dynamic optimization, stream processing**

I. INTRODUCTION

With recent advancements in big data, there has been a proliferation of stream processing systems (SPS) designed for data analysis [1], [2], [3], [4]. In these SPS, processing data with high throughput and low latency is critical for efficiently analyzing incoming data in real-time. In particular, it is essential to expedite the processing speed of queries on incoming data [5], [6]. When an SPS utilizes multiple devices to accelerate query processing, it becomes necessary to determine which device should execute each query. Efficiently addressing the device mapping problem is a crucial consideration to enhance the performance of the SPS. In practice, the latency for processing a specific query operator can vary across different devices. Each operator within a query may indeed exhibit a preference for certain devices. Additionally, the device preference of a query operator can sometimes change dynamically at runtime.

To tackle this challenge, numerous research efforts have emerged, broadly categorized into two types based on how queries are mapped to devices: at the query-level [7], [8], [9] and at the the operator-level [10], [11], [12]. However, regardless of the mapping level, most existing schemes are static. They either rely on GPU-specific libraries pre-bound to the GPUs [13] or use static mapping tables constructed based on predetermined device preferences [7], [10]. Static methods are not suitable for query mapping because factors such as input data size and the device to which the previous operator was mapped can affect the device preference of the current operator. This preference may vary at runtime. Consequently, static methods can negatively impact the runtime performance of an SPS [14], emphasizing the need for dynamically determining the device preference at runtime.

In this paper, we propose DYNO, a high-performance SPS that dynamically determines the optimal device mapping plan at the operator-level using an efficient tree-based machine learning (ML) model in a heterogeneous CPU-GPU environment. DYNO utilizes an ML algorithm to predict the execution time of each query operator based on real-time SPS execution logs, which include device type and data size. To maximize SPS performance, we introduce a novel turn-based collaborative updating method that facilitates simultaneous real-time training and inference of the model. In our turn-based updating algorithm, one teacher model is responsible for training the model, while two student models handle either conducting model inference or updates. This setup enables continuous real-time model training while simultaneously performing model inference with the improved quality of the trained model. Based on the prediction results from the student model, DYNO determines the operator-level device mapping plan for each query operator. Additionally, we have developed an efficient device mapping module that adapts the direct acyclic graph (DAG) shortest path algorithm to derive an optimal device mapping plan for a query in real-time. In the DAG, each node represents a distinct resource combination, which includes the computing resources allocated to each operator. The value of each node indicates the estimated execution time for the operator given its assigned resources. The first branch of operators in the query selects one of the nodes and

*Corresponding author.

sequentially identifies the path with the minimum sum of node values among the different paths leading to the final operator. We employ dynamic programming techniques to mitigate the computational complexity of the device mapping module.

We developed an efficient ML-based stream processing system implementation in the Spark 3.2.3 environment [15]. Experimental evaluations using various queries and traffic patterns from prior studies demonstrates significant performance improvements over existing state-of-the-art strategies for mapping operators to devices. DYNO achieved increased throughput and decreased latency compared to previous methods, leading to improved overall execution time. Furthermore, our optimized device mapping module finds the optimal device mapping plan in less than 1 millisecond. This efficiency allows us to completely hide the additional overhead using the prefetch method implemented in conjunction with the turn-based updating module. We also conducted extensive evaluations of DYNO to analyze the impact of various features on query execution. The main contributions of this paper are as follows:

- We identified that the device preference of each operator within a query changes dynamically at runtime.
- We compared and optimized various ML models to efficiently determine the optimized device mapping plan in a heterogeneous CPU-GPU environment.
- We proposed a novel turn-based updating algorithm that allows simultaneous model training and inference while ensuring low latency and high throughput.
- We implemented the proposed schemes in a real system and evaluated the performance using real world scenarios.

II. Background and Related Work

A. GPU-based Stream Processing Systems

SPS platforms, including Apache Flink [1], Apache Storm [2], Apache Spark [3], and Apache Samza [4] enable the processing and analysis of big data streams. [4]. These platforms decompose queries into multiple execution operators and execute them accordingly. Since data arrives continuously in real-time, achieving low latency and high throughput for query processing is even more critical than in typical batch processing environments. [5].

When adding a GPU to accelerate query processing, it is necessary to determine which device should execute the query. SPS converts the input query into an executable physical plan represented as a DAG of operators. To map these operators to the appropriate devices for execution, SPS can use either query-level or operator-level mapping. Query-level mapping executes the entire query on a single device, whereas operator-level mapping distributes the operators across multiple devices.

If an operator executes more quickly on the GPU than on the CPU, the GPU becomes the preferred device for that operator. This device preference is essential for accelerating query processing in a CPU-GPU environment, as the execution time is influenced by the device to which the operator is assigned. Research [16], [17], [18], [19] indicates that this choice directly affects overall processing speed.

TABLE I: Previous works on query mapping with CPU-GPU devices.

Previous Work	Mapping Level	Device Preference	Source Type	Planning Methodology
FineStream[10]	Operator	Static	Stream	Heuristic
SABER[9]	Query	Dynamic	Stream	Heuristic
GHive[8]	Query	Dynamic	Batch	Cost-based
GFlink[7]	Query	Static	Stream	Heuristic
HERO[12]	Operator	Dynamic	Batch	Cost-based
DYNO(ours)	Operator	Dynamic	Stream	ML

B. Related Work

In this section, we categorize query mapping strategies into static and dynamic methods based on how they consider the operator's device preference. Table I summarizes the characteristics of previous work on query mapping in a CPU-GPU environment.

Static Method. The most common strategy for mapping a query to a device is to consistently assign it to either the CPU or GPU [20]. This approach minimizes the overhead associated with transferring data between devices and converting data formats in a dedicated CPU-GPU environment. It can be particularly effective when data transfer overhead constitutes a significant portion of the total query processing time. FineStream [10] computed the device preference for each query operator through profiling in an integrated CPU-GPU environment. This approach initially considers the static device preference of an operator and performs operator-level mapping at runtime. GFlink [7] addresses the load balancing issue through locality-aware device mapping at the query level between GPU devices. These static methods pre-compute the device preference of the operators that comprise the query through profiling and maps the operators to devices based on these preference. FineStream leverages static device preferences to accelerate query processing by applying several runtime techniques, including light-weight resource reallocation.

Dynamic Method. This approach maps queries or operators to devices at runtime, taking into account that their device preferences can change due to various factors. GHive [8] is a notable study that addresses device preferences by using a univariate equation based on data size to map queries to either the CPU or GPU at runtime. HERO [12] employs linear approximation to estimate cardinality for operator-level device mapping. However, both GHive and HERO were applied in batch environments rather than streaming environments. In contrast, SABER[9] explores dynamic runtime mapping in a streaming context. SABER uses a query-level strategy that dynamically considers device preferences through throughput monitoring and maps the entire query to a single device. Nonetheless, FineStream's operator-level mapping has been shown to outperform SABER.

Limitation of Previous Methods. Existing research applying these methods has several limitations. First, while throughput and latency are crucial in streaming environments, solving the device mapping problem can introduce additional overhead.

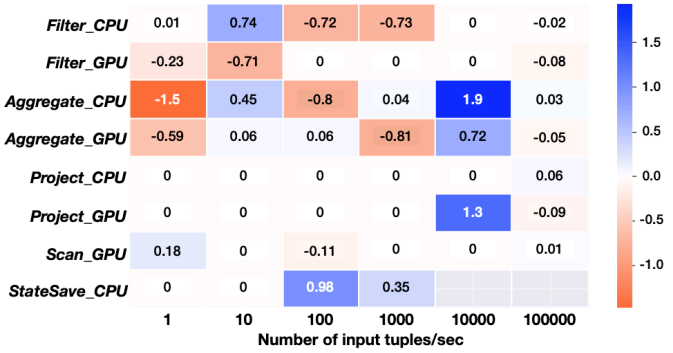


Fig. 1: Device preferences for $Exchange$ operators. The y-axis represents the previous operator and the device it was mapped to. A darker red color indicates faster execution on the GPU than CPU, and a darker blue color indicates faster execution on the CPU than GPU. How to calculate the value is shown in Equation 1. We represents up to two decimal places.

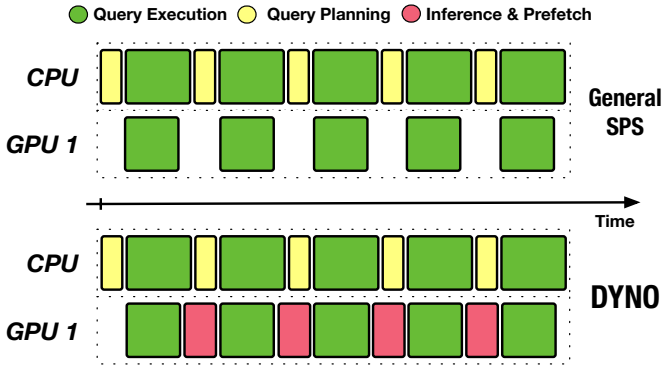


Fig. 2: The timeline when general SPS and DYNO process a query in a CPU-GPU environment. Timeline illustrates how query-processing tasks are assigned in a CPU-GPU environment in general SPS and DYNO. For clarity, we've simplified the entire process into a query planning phase and a query execution phase.

Thus, a strategy that accounts for this overhead is necessary. Second, static consideration of device preferences fails to address the changing dynamics of operators at runtime, making it insufficient for finding an optimal mapping scheme. Finally, traditional heuristic-based or cost-based planning methodologies often fall short of explaining the complex interplay of various factors in real-world environments. To find the optimal device mapping scheme, a method that dynamically considers the relationship between these factors is essential. These points are summarized in Table I.

C. Motivations

Dynamics of Device Preference. In order to identify the factors affecting the performance of SPS, we conduct experiments under the assumption that the device preference of each operator is dynamic in a streaming environment. We run our experiments using Spark-RAPIDS [13], one of the SPS that supports CPU and GPU operators, and use the Linear Road benchmark [21] for input raw data. In this experiment, we use only one CPU and one GPU, both configured identically to those used in our evaluations in Section IV. The experiments vary in terms of the type of previous operator, the device to which the previous operator was mapped, the number of tuples of raw data fed into SPS in real-time, and the exponentially increasing size of the fed data to measure dynamics.

Figure 1 shows a heatmap of the operator's device preference. As described in Equation 1, if the device preference ($dev\_preference$) is less than 0, it indicates that the execution time on the GPU ($time_{GPU}$) is less than the execution time on the CPU ($time_{CPU}$). Therefore, when $dev\_preference$ has a negative value, there is a higher preference for the GPU.

$$dev\_preference = log(\frac{time_{GPU}}{time_{CPU}}) \quad (1)$$

As shown in Figure 1, there are dynamics due to several factors in the case of $Exchange$. For example, in the case of $Filter\_GPU$, when the number of tuples in the input raw data is 10, Exchange's device preference value is -0.71, which means that the execution time on GPU is about 5 times faster than when running on CPU. However, in the case of $Filter\_CPU$, under the same condition, the device preference value is 0. In other words, even if the previous operator is the same, the difference in device preference occurs depending on which device the previous operator was executed on. In addition to this observation, in the case of $Aggregate\_CPU$, we can see that various factors cause dynamics, such as the number of tuples in the input raw data causes the device preference to vary.

Since these various factors change dynamically at run time, finding an optimal mapping scheme is challenging. Furthermore, simple heuristic methods or mathematical models that are not machine learning algorithms are often ineffective in accurately reflecting the complex correlations between factors.

Idleness of GPU in Planning Phase. In conventional SPS, GPU resources are not utilized in the planning phase as shown in Figure 2. Also, additional operations (e.g., writing the results to a file) occur on the CPU after the query processing is finished, so the GPU idle period is repeated for every processing. We focused on this GPU idle period in order to find the exact optimal plan and completely eliminate the overhead it causes. As mentioned above, we need to consider all the complex correlations between the operators to get the exact optimal plan. However, if this process is computed sequentially in the planning phase, the additional overhead can actually degrade the overall performance.

Therefore, we propose a new scheme to determine an optimal mapping plan using a machine learning algorithm, known for its efficiency in learning these correlations. In addition, as shown in Figure 2, the overhead of applying the machine learning algorithm was completely eliminated by overlapping the idle sections of the existing GPU.

III. Design and Implementation

A. Design Challenges

Over the past few years, ML has regained significant interest across various domains as a means to address domain-specific problems. For example, in the field of query processing, there has been substantial research on ML models developed to predict query execution times [22], [23], [24], [25]. Most previous work has focused on non-stream processing, where data processing latency is relatively less critical. However, in streaming environments, where both throughput and latency are crucial, applying ML models to develop SPS presents several challenges.

Challenge 1. To minimize processing time in SPS, it is necessary to map queries to the optimal device. To achieve this, an optimal ML model that satisfies high-accuracy with low-latency needs to be applied in SPS. Therefore, it is required to select a model by comparing the training and inference latency as well as accuracy across diverse ML models.

Challenge 2. While processing real-time incoming data in SPS, inferring the execution time of the operator during the planning stage accounts for a significant portion of the overall processing time. Hence, minimizing the inference time of the model is essential to maximize performance.

Challenge 3. As training data is fed into the SPS in real-time, a smaller amount of data can be used to initially train the prediction model. Therefore, to achieve a higher quality ML-based prediction model, continuous training of the model with an increased size of data is required. Moreover, it is crucial to employ the most recently updated model for inference with minimal overhead during the planning process.

As described above, there are several challenges associated with applying ML models to stream query processing. To efficiently address aforementioned challenges and produce the optimal device mapping plan, we propose our new DynO system, which is the first to leverage ML models for device mapping optimization on SPS.

B. Overview of DynO

In this sub-section, we present a high-level overview of our DynO system. Figure 3 shows the overall workflow of DynO, which consists of the *Turn-based Updating Module* and the *Device Mapping Module*. The *Turn-based Updating Module* employs two GPU devices to concurrently perform training and inference of an ML-based regression model. This model predicts the execution time of operators across various devices. Then the *Device Mapping Module* finds the optimal device mapping plan using the predicted execution time from the *Turn-based Updating Module* to minimize latency within the stream query processing systems. These two modules communicate dynamically with each other via a socket, and they are ported to the existing stream processing systems.

In the *Turn-based Updating Module*, we develop a novel turn-based updating scheme that exploits an efficient tree-based ML model to reduce latency while improving model accuracy. As shown in Figure 3, our turn-based updating

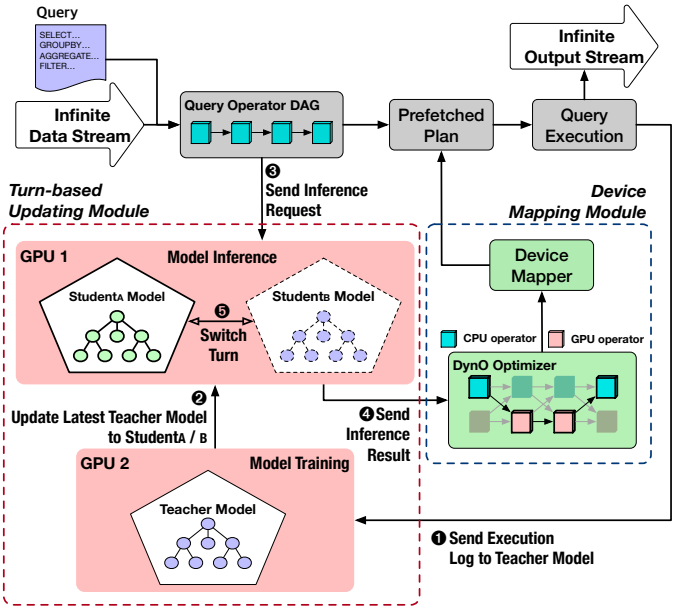


Fig. 3: Overview of DynO. DynO consists of two main modules: Turn-based Updating Module and Device Mapping Module. $Student_A$, represented by the bold line, is responsible for inference. $Student_B$, represented by the dotted line, is responsible for model overwrite. After the overwrite is complete, $Student_A$ and $Student_B$'s roles are reversed and it becomes Turn B. The Device Mapping Module uses bold and light arrows to separate selected operators. DynO prefetches the optimized plan to SPS in advance so that it is ready for use in the next processing.

scheme basically consists of one $Teacher$ model for model training on GPU 2, and two Student models (i.e., $Student_A$ and $Student_B$) for model inference on GPU 1. While GPU 2 continuously trains the model using an increased size of the real-time incoming training dataset, GPU 1 is involved in model inference. The $Teacher$ model continuously trains the tree-based regression model using the real-time incoming dataset. On the other hand, each of two $Student$ models either performs inference using the most recently trained model or updates the model from the $Teacher$ model. For example, upon completion of query execution, ❶ GPU 2 receives the execution logs and trains the $Teacher$ model with new incoming data. ❷ After training, the most recently trained $Teacher$ model is sent to GPU 1 to update the $Student_B$ model. During the update of the $Student_B$ model, ❸ when the Query Operator sends the inference request, ❹ GPU 1 executes inference using the latest $Student_A$ model, previously copied from the trained $Teacher$ model on GPU 2, and sends the predicted value to the *Device Mapping Module*. That is, while updating the $Student_B$ model, $Student_A$ model is concurrently used for model inference on GPU 1. ❺ After the completion of updating the $Student_B$ model, it is used for model inference, while $Student_A$ begins updating based on the latest trained $Teacher$ model on GPU 2. Therefore, by

effectively alternating the turns of $Student_A$ and $Student_B$ models on the same GPU 1, it is feasible to conduct model inference and model updating concurrently. This approach minimizes latency while enhancing model accuracy by leveraging the latest trained model from the $Teacher$ model.

Using the inference results from the *Turn-based Updating Module*, the *Device Mapping Module* determines the optimal device mapping plan for the query operator and executes the selected plan to generate additional execution logs. In the *Device Mapping Module*, we solve the problem of dynamics by organizing the execution time of operators into a DAG-like data structure to find the shortest path. The optimal plan is then prefetched and ready for use in the next processing. This prefetching process allows us to hide the overhead caused by the two modules. The overall process of DYNO is recursively performed in real-time.

C. Turn-based Updating Module

The *Turn-based Updating Module* receives a query operator DAG of CPU operators from the SPS. The module then preprocesses the DAG to generate the appropriate inference requests to estimate the execution time of the operators. These requests are passed to the model in DYNO. In this sub-section, we describe the details of an ML-based regression model used in the turn-based updating module.

Tree-based Regression Model. There are various types of ML-based regression models, including tree-based models and deep neural networks (DNNs). Especially for tabular and categorical data, it is reported that tree-based machine learning models outperform DNNs in terms of model accuracy [26], [27], [28], [29]. Moreover, the computational complexity of tree-based machine learning models is much lower than that of DNNs-based models. Therefore, we employ a special type of tree-based regression model, gradient boosting [30], to precisely and efficiently predict the runtime of the operators in the *Turn-based Updating Module*.

$$\begin{aligned} F_m(x) &= F_{m-1}(x) + \gamma_m h_m(x) \\ \gamma_m &= \arg\min_{\gamma} \sum_{n}^{i=1} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)) \end{aligned} \tag{2}$$

Equation 2 formulates the gradient boosting algorithm, where γ_m, L, $h_m(x)$, m, and $F_m(x)$ denote multiplier, loss function, base learner (i.e. tree), the mth iteration of training, and the model, respectively. Based on the input features listed in Table II, our DYNO trains the gradient boosting model to accurately predict the execution time of the current operator. The primary input features we used include input data size, current operator, current device, previous operator and previous device. When finding the optimal device mapping plan based on the predicted result from the gradient boosting algorithm, the device preference for an operator depends on the device to which the previous operator was mapped. Therefore, to consider the dynamics of device preference, we incorporate major factors affecting device preference as input for the prediction model.

Collaborative Turn-based Updating Method. As shown in Figure 3, in the *Turn-based Updating Module*, a gradient boosting based $Teacher$ model is trained on GPU 2, while model inference is performed on GPU 1 in parallel. Algorithms 1 and 2 provide pseudo-code for describing how the $Teacher$ model on GPU 2 and $Student_A$ model on GPU 1 collaborate. Note that the $Student_B$ model on GPU 1 is used in the same manner as the $Student_A$ model. The only difference lies in the order of operations. In addition, both the $Student_A$ and $Student_B$ models maintained on GPU 1 are processed simultaneously. For example, when GPU 1 receives an inference request from a query operator, it utilizes $Student_A$ to conduct model inference and sends the inference result to the *Device Mapping Module*. At the same time, GPU 1 receives the latest trained $Teacher$ model from GPU 2 to copy the updated parameters to the $Student_B$ model. Meanwhile, GPU 2 constantly trains the $Teacher$ model using real-time incoming execution logs, which include data such as the actual query execution time and the devices used for previous and current operators. To send the trained weight parameters of $Teacher$ model to the $Student$ model on GPU 1, GPU 2 uploads the parameters to a global variable, as described in line 10 of Algorithm 1. After the uploading process is completed, the model parameters of either $Student_A$ or $Student_B$ are replaced by the latest parameters that have been uploaded. One of the $Student$ models, which has not been updated, is used for performing model inference based on the previously uploaded parameters. Algorithm 1 and 2 operate in a turn-based update manner until the SPS terminates. Throughout the SPS execution, these algorithms remain active and are

TABLE II: Input features of tree-based regression model in turn-based updating module.

Features	Description	Data type
$inputRow$	Number of tuples per processing	Float
$currOp$	Current operator name	String
$prevOp$	Previous operator name	String
$currDev$	Whether the current operator run on the CPU or GPU	Boolean
$prevDev$	Whether the previous operator run on the CPU or GPU	Boolean
$opLoc$	Location of the node to distinguish same operators in a query	String

Algorithm 1 Teacher Model in Turn-Based Updating Module

// $weightFlag$: If true, the latest model has been uploaded
// $globalModel$: Latest model of teacher model
global $weightFlag, globalModel$

```
1: procedure TEACHER
2:     while True do
3:         get execution log from Spark
4:         preprocess execution log
5:         if model is not created then
6:             create and compile model
7:         end if
8:         train model by execution log
9:         if cycle is reached then
10:            globalModel ← latest updated model
11:            weightFlag ← True
12:        end if
13:    end while
14: end procedure
```

Algorithm 2 Student Model ($Student_A$) in Turn-Based Updating Module

// $finishFlag$: If true, inference is done
// $workingID$: Thread ID of responsible for inference
global $finishFlag, workingID, weightFlag, globalModel$

procedure STUDENTA
  create and compile model
  $myID \leftarrow$ this thread Id
  while $True$ **do**
    $finishFlag \leftarrow False$
    while $workingID = myID$ **do**
      get preprocessed DAG
      if the model needs cold start **then**
        $result \leftarrow$ random value
      else
        $result \leftarrow$ prediction of model
      end if
      send $result$
      if $workingID \neq myID$ **then**
        $finishFlag \leftarrow True$
      end if
    end while
    wait for $weightFlag$ set
    overwrite model to $globalModel$
    $weightFlag \leftarrow False$, $workingID \leftarrow myID$
    wait for $finishFlag$ set
  end while
end procedure

responsible for training and inferring the ML model. In our DYNO implementation, we use appropriate locks on the global variables used in each model to prevent race conditions.

Algorithm 3 DAG Shortest Path Algorithm

Input : Estimated execution time DAG, $estiExec$
Output : Route of shortest path, $shortPath[idx]$

procedure FINDSHORTESTPATH($estiExec$)
  initialize $path_0, shortPath$
  $prev \leftarrow \{0:(0,2),\ 1:(0,2),\ 2:(1,3),\ 3:(1,3)\}$
  $id \leftarrow 1$, $maxID \leftarrow$ the number of operators
  while $id \leq maxID$ **do**
    for $i \leftarrow 0$ to 3 **do**
      $rPrev = path_{id-1}[prev[i][0]]$
      $lPrev = path_{id-1}[prev[i][1]]$
      $path_{id}[i] \leftarrow min(rPrev, lPrev)$
      $path_{id}[i] = path_{id}[i] + estiExec_{id}[i]$
      update $shortPath[i]$
    end for
    $id \leftarrow id + 1$
  end while
  $idx \leftarrow$ index of minimum in $path_{maxID}$
  return $shortPath[idx]$
end procedure

D. Device Mapping Module

Given the predicted execution time of each operator from the *Turn-based Updating Module*, the goal of the *Device Mapping Module* is to identify the optimal device mapping plan using the DAG shortest path algorithm. In the DAG algorithm, to find the shortest path from one node to another, the first step is to topologically sort each node and then expand from the starting node to its neighbors one by one. However, in the case of SPS, the operator nodes are already topologically sorted because there exists a dependency

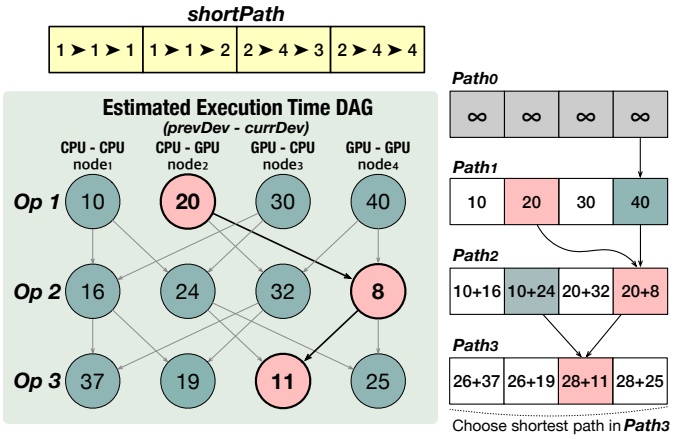


Fig. 4: Flow of DAG shortest path algorithm. $Path_i$ indicates the shortest distance to reach each node from the first operator to the i+1st operator. The node colored in red is the shortest path.

between each operator that constitutes the query, and the tree-like data structure starts from the operator that needs to be executed last. In addition, when representing the dynamics of computing resources caused by the previous device as (previous device, current device) pairs, there are a total of four different types: (CPU, CPU), (GPU, CPU), (CPU, GPU), and (GPU, GPU). In other words, if the execution time calculated by considering the dynamics of all operators that make up the query is expressed in a DAG structure, there are four nodes for each operator, and each node is connected to the appropriate previous operator. If each node has the execution time predicted by the *Turn-based Updating Module* as a value, the shortest path is able to connect the first operator node and the last operator node in such a DAG becomes the optimal device mapping plan.

Algorithm 3 presents the pseudo-code, while Figure 4 shows the flow of the DAG shortest path algorithm in the *Device Mapping Module*. First, the graph consists of nodes (i.e., operators) with the predicted execution time of each operator as the value of the node. Using the example in Figure 4, a total of three operators are used, and each operator is represented by one layer. Each layer consists of four nodes, with each node representing the dynamics of (CPU, CPU), (CPU, GPU), (GPU, CPU), and (GPU, GPU). Arrows point to the next node that can be logically mapped from each node. The problem of finding an optimal device mapping plan can be reformulated as the problem of finding a path from the top layer through all layers to the bottom layer, such that the sum of the values of the nodes traversed in the DAG graph is minimized. To efficiently solve this problem, we adapt a dynamic programming technique. As shown in Figure 4, the array $Path$, consisting of four elements, represents fours nodes. Each element stores the length of the shortest path that can reach that node at the current operator. We additionally use the array $shortPath$, to store the route of the shortest path to

TABLE III: Query details of real-world streaming workloads used in evaluation. We categorized and selected three queries based on the characteristics of each query to verify that DYNO performs well even in environments where it is difficult to learn due to the large variation caused by the some features proposed in Table II, as we labeled Related Features.

Query	Description	Related Features
Q_1	SELECT L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment FROM SegSpeedStr (range 30 slide 1) as A, SegSpeedStr as L WHERE (A.vehicle == L.vehicle)	$inputRow$, $opLoc$
Q_2	SELECT timestamp, highway, direction, segment, AVG(speed) as avgSpeed FROM SegSpeedStr (range 300 slide 1) GROUPBY (highway, direction, segment) HAVING (avgSpeed $\leq$ 40.0)	$currOp$, $prevOp$
Q_3	SELECT timestamp, highway, direction, segment, COUNT(vehicle) as numVehicle FROM SegSpeedStr (range 30 slide 1) GROUPBY (highway, direction, segment)	$currDev$, $prevDev$, $opLoc$

each node. Consider the case of $Path_3[2]$ shown in Figure 4, where the algorithm navigates from the second layer to the third layer. To reach $Path_3[2]$, the algorithm is required to pass through either $Path_2[1]$ or $Path_2[3]$. Therefore, the algorithm selects the smaller of the two values and adds it to the value of the $node_4$ in operator 3, storing the result in $Path_3[2]$. When the last layer is reached, the algorithm selects the smallest of the four values, which becomes the final shortest path. For example, in Figure 4, the selected shortest path is $node_2 \rightarrow node_4 \rightarrow node_3$, indicating that our *Device Mapping Module* chooses to map each operator from GPU $\rightarrow$ GPU $\rightarrow$ CPU. Using our approach, the shortest path finding problem can be solved in $O(N)$ time complexity, where N is the number of operators that need to be mapped to the device.

IV. EVALUATION

A. Evaluation Setup

Configurations. All of our experiments were conducted on a single machine equipped with one Intel i7-13700F 16-core 2.1 GHz CPU with 32GB of memory and two NVIDIA RTX 3070 GPUs, each with 8GB of GPU device memory. We used one master and two workers, each running a single executor, to demonstrate the superiority of DYNO. Since DYNO does not introduce any additional overhead when applied to existing SPS, we expect it to perform better with more nodes. In this experiment, we aim to verify the superiority of DYNO using a minimal number of nodes. All processes used for two $Student$ models, one $Teacher$ model, and Spark were executed concurrently. For the Spark setting, we used the default values provided.

Comparisons. We evaluated the processing performance of All-CPU, All-GPU, FineStream (e.g., using static mapping) [10], and DYNO from various perspectives in a CPU-GPU heterogeneous environment. FineStream maps queries at the operator-level, considering each operator's device preference in the streaming environment. This approach assumes that each operator has a static device preference for either CPU or GPU. Based on the operator's device preference table provided by FineStream, we built a static-mapping strategy and compared its performance to that of DYNO.

Workloads and Stream Traffic Types. In this experiment, we utilized real-world streaming workloads from the Linear Road Benchmark (LRB) [21], which is commonly used to evaluate the performance of SPS in recent studies [9], [10]. The LRB uses a diverse set of query, including join, project, aggregate, group-by, and select. We select three continuous queries in LRB, labeled Q_1, Q_2, and Q_3, as described in Table III. Q_1 contains a join, so the processing time is strongly affected by the $inputRow$. Q_2 contains all other types of operators except join, so the variation between $currOp$ and $prevOp$ is large. Q_3 has the largest number of operators, which makes the number of possible combinations of $currDev$ and $prevDev$ large.

Apache Kafka [31] served as the message broker, with Spark acting as the stream processing engine, enabling us to replicate the entire streaming system. To assess each framework's performance under various real-world conditions, we categorized the traffic scenarios as outlined in the subsequent table. We adjusted the workload size to what our experimental setup could manage within Spark's default configuration and machine configuration, aiming to avoid capacity problem.

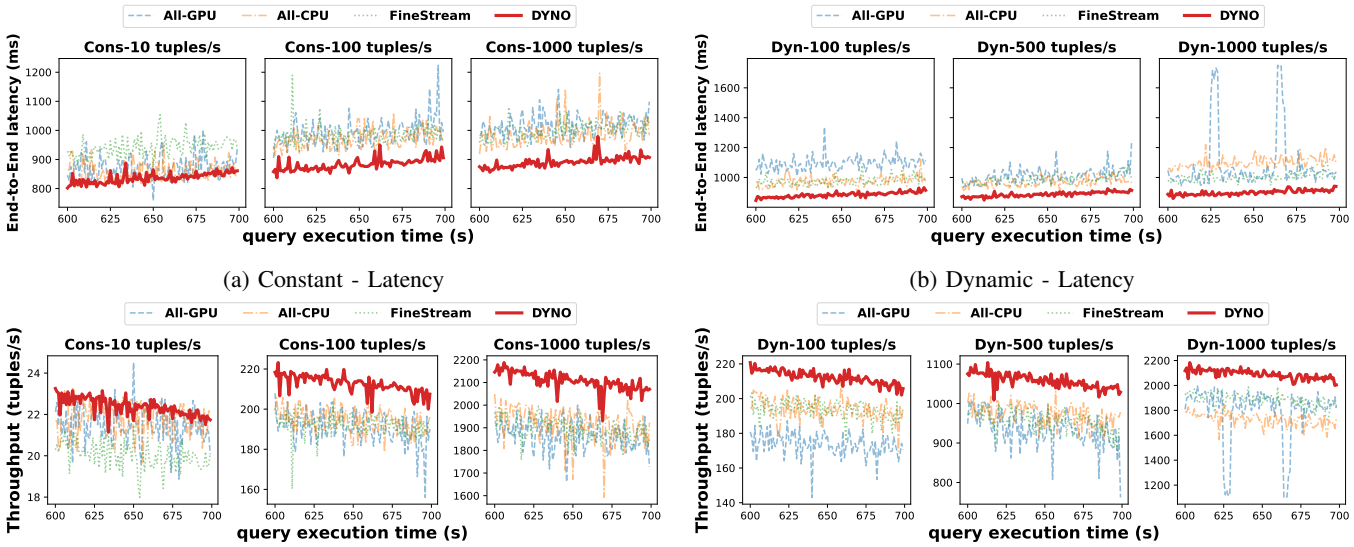
- **Constant Traffic**: A set number of tuples are input every second. For Q_1, we experiment with constant traffic of 10, 50, and 100, and for Q_2 and Q_3, we experiment with constant traffic of 10, 100, and 1000.
- **Dynamic Traffic**: In the real world, input traffic fluctuates rather than remaining constant. To demonstrate that our DYNO performs efficiently under fluctuating traffic as well as constant traffic, we also evaluated it using dynamic traffic. Every second, a random number of data tuples are ingested and form a dataset. These random numbers are generated every second, achieving a normal distribution of each random number in Table IV as an average point.

TABLE IV: Types and traffics.

Types	Traffics (Number of Input Tuples per second)
Constant	10, 100, 1000
Dynamic	Follow a normal distribution with 100, 500, 1000 as the mean.

* **Because of the capacity problem by default configure, we test Q_1 under 10, 50, 100 of constant traffic and dynamic traffic. The data size of a tuple is fixed at about 0.1 KB.**

** **We abbreviated type and traffic as Type-Traffic (e.g., Cons-10 and Dyn-1000) in the figures presented in this section.**



(a) Constant - Latency

(b) Dynamic - Latency

(c) Constant - Throughput

(d) Dynamic - Throughput

Fig. 5: Graphs showing the performance (i.e. throughput, latency) of DYNO with various traffic types after initial training. High peaks in Fig. 5d, 5b caused by dynamic input traffic which occasionally exceed the GPU's capacity. DYNO outperformed the traditional method on all queries, so we plotted all graphs for Q_2 as a representative to further analyze how DYNO's method performs on different traffic. The lower the latency graph and the higher the throughput graph, the better the performance. DYNO shows high performance as well as less volatility than traditional methods, resulting in more stable processing.

B. Performance

We conducted experiments to validate how DYNO performs compared to traditional methods under different traffic type conditions, as outlined in Table IV. Our method proposes a way to use ML models to accurately determine an operator's device preference in real-time, and can be integrated into existing research on monitoring or predicting device preference. To compare the difference between predicting device preference in real-time and statically computing it, we leveraged the device preference table proposed by FineStream [10], the state-of-the-art in operator-level mapping.

Figure 5 shows the performance comparison of the experiment. It is obvious that for all cases of traffic and type, our DYNO shows a significant performance improvement. Furthermore, DYNO handles performance with less oscillations and spikes and is more stable than the other three mapping schemes (i.e. All-GPU, All-CPU, and FineStream). In terms of latency, the three compared mapping schemes appear to oscillate more for constant traffic than for dynamic traffic. However, this observation is influenced by the range of the y-axis. Upon considering the full range, we find that this is not the case. We can see that even at relatively very low traffic (i.e. 10 tuples/s), DYNO ensures the best possible throughput and latency. These results show that DYNO works effectively even though the percentage of query processing time that can be reduced by the optimal plan generated by DYNO is very small in the end-to-end latency. As our DYNO completely hides the additional overhead through the prefetch method while generating optimized mapping plans in real-time, it consistently exhibits fast processing time.

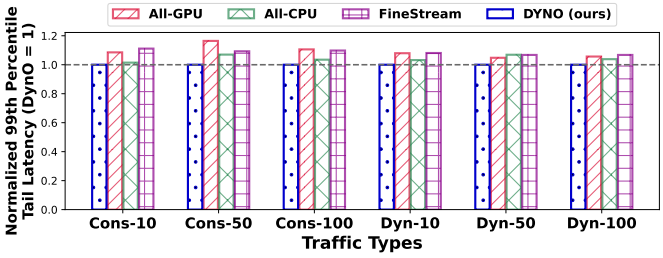
We also measured tail latency to compare the performance of DYNO with other approaches. Figure 6 illustrates the tail latency of 99^{th} percentile under both constant and dynamic traffic conditions for each query described in Table III. In general, the tail latency reduction effect seems to be better as the input rate of traffic increases, especially for Q_3. In addition, DYNO showed a significant tail latency reduction effect for both constant and dynamic traffic. This result indicates that DYNO effectively optimized the overhead incurred by adding the learning and inference process of ML models in real-time processing, and at the same time, the optimal plan is effective.

C. Device Mapping Overhead Analysis

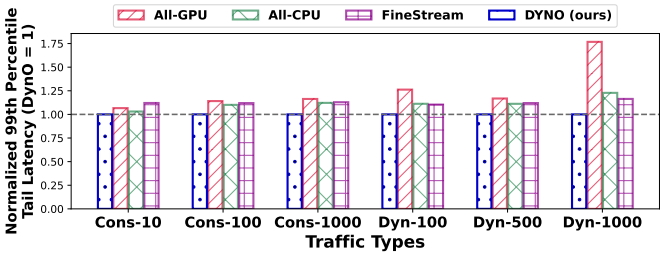
In this sub-section, we analyzed the latencies optimized by DYNO by measuring the actual latency. Table V shows the average latency for each query for three additional types of latency that occur as a result of applying the ML model. Each latency is defined as follows.

- **Mapping latency** Once the optimal device mapping plan is determined, each task is mapped to a device before it is actually executed, which is how long it takes.
- **Communicate latency** This latency refers to the time spent purely communicating with the model, excluding the model's inference time.
- **Path latency** The time it takes to actually perform the DAG shortest path algorithm and predict the optimal path.

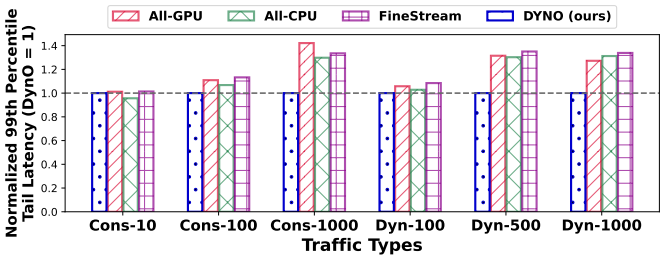
The difference in mapping latency is noticeable in Table V. This is because the mapping latency is the process of mapping all the child functions included in the actual execution of



(a) Q_1, 99^{th} Percentile



(b) Q_2, 99^{th} Percentile



(c) Q_3, 99^{th} Percentile

Fig. 6: 99^{th} percentile tail latency of each query and traffic type. Every values are normalized by the DYNO's value of each case. This is the result of processing the query from the left using the DYNO, All-GPU, All-CPU, and FineStream(using static table) mapping methods.

TABLE V: Three types latency of DYNO for Q_1, Q_2 and Q_3.

Query	Mapping Latency	Communicate Latency	Path Latency
Q_1	0.07 ms	0.09 ms	0.11 ms
Q_2	0.06 ms	0.09 ms	0.12 ms
Q_3	0.27 ms	0.10 ms	0.13 ms

each operator, and Q_3 has about 4 times more child functions than Q_1 and Q_2. Therefore, the mapping latency for Q_3 is about 4 times larger than other queries. The path latency and communicate latency remain the same as the number of operators increases (i.e., the number of operators is less in the order of Q_1, Q_2, and Q_3). Moreover, the sum of these three types of device mapping overhead is less than 1% of the end-to-end processing time according to our experiments. This result shows that the DAG shortest path algorithm applied by DYNO can quickly communicate with the ML model and find the optimal path even as the number of operators increases.

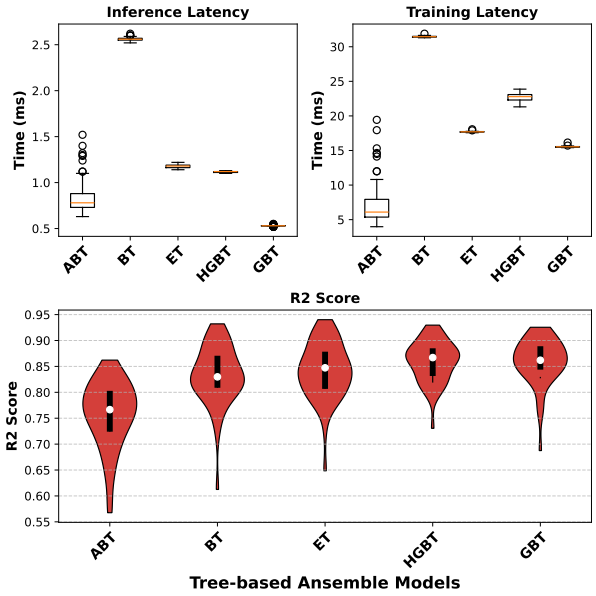


Fig. 7: Comparison of several variants of tree-based ML models in terms of inference latency, training latency, and R^2 score [32]. In box plots, the values represented by the circles are the fliers. The violin plot below show the R^2 score distribution for each model. In the violin plot, the black boxes represent the range of the 1^{st} and 3^{rd} percentile values, and the white circle is the mean value.

D. Comparison of Tree-based ML Models

In stream processing, it is very important to process infinitely incoming data with high throughput and low latency at the same time. Therefore, when applying ML models, it is necessary to minimize the additional latency. Simultaneously, high accuracy of the model must be guaranteed so that an optimal plan can be found based on it. Therefore, in this subsection, we compared the latency and accuracy of various tree-based ML models when applied to the DYNO environment.

Figure 7 shows a comparative analysis of Inference Latency (IL), Training Latency (TL), and model accuracy across a test dataset. We used the same LRB dataset used in the DYNO performance validation experiments to evaluate tree-based models. The number of features and volume were also measured under the same environment as DYNO. Specifically, the features were applied as shown in Table II. Additionally, we used 10-fold cross validation to evaluate the model's performance. The total size of the dataset, including the test data, was approximately 20,000 entries.

We utilize five eminent tree-based ML models as benchmarks, including AdaBoostTree (ABT) [33], BaggingTree (BT) [34], ExtraTree (ET) [35], HistGradientBoostTree (HGBT), and GradientBoostTree (GBT) [36], [30], [26], all sourced from the scikit-learn library. Each model employs a decision tree with a maximum depth of three as the foundational learner. The dataset was partitioned randomly into a 10% test set and a 90% training set for each experimental iteration,

with 50 cycles of training and inference executed. According to the experimental results, in the case of ABT, both TL and IL are low, but there are more fliers than other models such as BT and ET, and the average accuracy is below 0.8. In the case of GBT, IL is the lowest, and the average accuracy is the highest at over 0.85. However, its TL is about 15ms on average, which is higher than ABT. The training process of an ML model always includes an inference step and a weight modification step, so TL has a very large value compared to IL. Moreover, in an environment where training and inference are performed in real time, TL should be prioritized over IL for low latency. However, DYNO uses a turn-based update module to hide TL and minimize IL. In the case of GBT, IL is up to 5 times faster in the DYNO environment compared to other models and has high accuracy. Therefore, we fine-tuned the GBT model to ensure low latency and high accuracy and applied it to DYNO.

E. Analysis of Feature Importance

Unlike other studies, we used a variety of features for learning to predict the operator's device preference. In this sub-section, we analyzed how each feature affects the learning according to the query type through Mean Decrease Impurity (MDI) and Permutation Importance (PI), and showed that the use of these features is valid. MDI and PI are both metrics used in many recent studies to measure feature importance in tree-based models [37], [38], [39]. MDI is based on the decrease in impurity that occurs when each feature is used for splitting in the tree. Therefore, MDI provides a predictive impact for each input feature [40]. On the other hand, PI measures the decrease in model accuracy when the values of the features are randomly shuffled. Therefore, PI provides the importance of each feature in predicting the target [41].

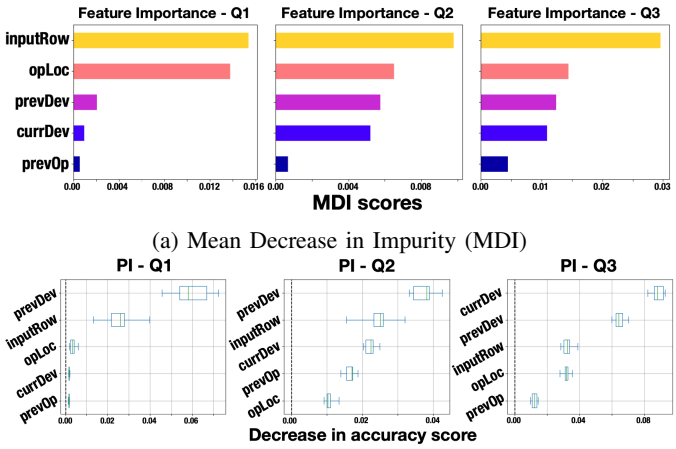
$$Imp(X_m) = \frac{1}{N_T} \sum_{T} \sum_{t \in T : v(s_t) = X_m} p(t) \Delta i(s_t, t) \quad (3)$$

Equation 3 shows the definition of MDI. MDI indicates the importance of a variable X_m. $p(t)\Delta i(s_t, t)$ is weighted impurty decrease where $p(t)$ is the proportion N_t/N of samples reaching the node t. $i(t)$ is impurity function, and $v(s_t)$ is the variable used in split s_t.

$$PI_i(F^j) = OOBError'_i - OOBError_i \quad (4)$$

$$PI(F^j) = \frac{1}{c} \sum_{i=1}^{c} PI_i(F^j) \quad (5)$$

Equation 4 shows how to calculate PI of feature F^j based on ith tree. $OOBError_i$ is Out-Of-Bag (OOB) error of ith tree, and $OOBError'_i$ is OOB error of new dataset which the values of feature F^j are randomly rearranged. We can obtain final PI value of feature F^j by averaging the PI values of c trees. Both MDI and PI indicate that the larger the value, the greater the influence of the feature on the model. However, MDI indicates the influence of each feature on the construction of decision trees, and PI indicates the influence of each feature on the accuracy of the model. Therefore, to analyze the influence of the input features we proposed, it is required to consider both MDI and PI.



(a) Mean Decrease in Impurity (MDI)

(b) Permutation Importance (PI)

Fig. 8: MDI scores and PI for input features. Since the information in $currOp$ is essential for predicting the execution time of an operator, we show the values except for $currOp$.

Figures 8a and 8b show the results of measuring MDI and PI under dynamic traffic for three queries each. First, in Figure 8a, $inputRow$ and $opLoc$ have high importance for all queries, but $opLoc$ has about twice as much value in Q_1 and Q_3 compared to Q_2. This is because there are fewer duplicate operators in the operator plan in Q_2. Without duplicate operators, the importance of $opLoc$ is relatively low because the $currOp$ information is sufficient to specify which operator the model is. MDI also overestimates high-cardinality features, and therefore the fact that the importance of high-cardinality $opLoc$ and low-cardinality $prevDev$ and $currDev$ were calculated similarly in Q_2 and Q_3 indicates that $prevDev$ and $currDev$ have a greater practical impact than $opLoc$ as shown in Figure 8b. Figure 8b shows the PI in the form of a boxplot. As mentioned earlier, $currDev$ and $prevDev$ have higher importance compared to $opLoc$ in Q_2 and Q_3. Furthermore, $inputRow$ shows a similar level of PI in all kinds of queries, while other features show different ranges in different queries, indicating that the features that mainly influence the operator's device preference vary depending on the type and form of query. Based on the analysis in Figures 8a and 8b, it is obvious that the features that affect the prediction of execution time change dynamically depending on the nature of the query. Therefore, it is crucial to consider all of them to make an accurate prediction.

V. DISCUSSION

Our study highlights that when processing stream data, the tree-based model effectively predicts the execution time of each operator and can simultaneously learn and infer in real-time without any prior information. In this section, we discuss two critical aspects of DYNO necessary for its effectiveness

in a real-world setting: the scalability of DYNO and the additional device cost of training the model.

A. Scalability of DYNO

We used one CPU and two GPUs to validate the performance of DYNO in this study. However, we envision that DYNO can be effectively applied to environments with multiple CPUs and GPUs. This is because DYNO learns about the devices in the execution environment without any prior information and maps each operator to the optimal device. Additionally, since the tree-based model does not require encoding of learning features, it can be easily extended to handle multiple CPUs and GPUs. Furthermore, in multi-node configurations with multiple GPUs, developing an efficient scheduler with priority queues for each multi-GPU and multi-CPU node will enhance DYNO's scalability.

B. Additional Cost for Model Training

In a streaming environment with continuous data ingestion, both training and inference occur simultaneously, leading to high GPU utilization. Although DYNO supports training and inference on the same GPU, its design prioritizes performance by separating GPUs for these tasks, which aligns with the streaming system's focus on low latency. We overlap DYNO's inference phase with GPU idle periods in the existing SPS to improve overall GPU utilization. In addition, DYNO uses a tree-based model, which requires fewer computing resources for training and inference, and the resource demands remain relatively stable even as the cluster size grows. Consequently, the additional cost of model training is minimal, and its significance decreases as the cluster size increases.

VI. Conclusion

In this paper, we propose DYNO, a new ML-based SPS that optimizes the device mapping of query operators in a dedicated CPU-GPU environment in real-time. DYNO introduces an innovative mapping technique suitable for any SPS processing queries built with operators, addressing the device mapping problem tackled in previous studies. Moreover, DYNO is designed to enhance performance on various SPS platforms like Flink, Spark, which also manage operator-based queries. Evaluations from various perspectives demonstrate that the proposed approach of applying ML to real-time optimization maintains high throughput and low latency, effectively enhancing overall performance with high efficiency.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386) (No. NRF-RS-2024-00359076) (No. RS-2024-00416666).

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[2] M. H. Iqbal, T. R. Soomro, *et al.*, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[3] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "Mllib: Machine learning in apache spark," *The journal of machine learning research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[4] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[5] N. Bhatt and A. Thakkar, "An efficient approach for low latency processing in stream data," *PeerJ Computer Science*, vol. 7, p. e426, 2021.

[6] S. Wu, M. Liu, S. Ibrahim, H. Jin, L. Gu, F. Chen, and Z. Liu, "Turbostream: Towards low-latency data stream processing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 983–993, IEEE, 2018.

[7] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1275–1288, 2018.

[8] H. Liu, B. Tang, J. Zhang, Y. Deng, X. Yan, X. Zheng, Q. Shen, D. Zeng, Z. Mao, C. Zhang, *et al.*, "Ghive: accelerating analytical query processing in apache hive via cpu-gpu heterogeneous computing," in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 158–172, 2022.

[9] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 555–569, 2016.

[10] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "Finestreamfine-grained windowbased stream processing on cpugpu integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 633–647, 2020.

[11] J. Liu, F. Zhang, H. Li, D. Wang, W. Wan, X. Fang, J. Zhai, and X. Du, "Exploring query processing on cpu-gpu integrated edge device," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4057–4070, 2022.

[12] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 733–744, 2017.

[13] NVIDIA. https://github.com/NVIDIA/spark-rapids, 2021.

[14] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl, "Grizzly: Efficient stream processing through adaptive query compilation," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2487–2503, 2020.

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[16] V. W.-z. Yu and M. Govoni, "Gpu acceleration of large-scale full-frequency gw calculations," *Journal of Chemical Theory and Computation*, vol. 18, no. 8, pp. 4690–4707, 2022.

[17] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pp. 1617–1632, 2020.

[18] V. Rosenfeld, S. Breß, and V. Markl, "Query processing on heterogeneous cpu/gpu systems," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–38, 2022.

[19] M. Nelson, Z. Sorenson, J. M. Myre, J. Sawin, and D. Chiu, "Parallel acceleration of cpu and gpu range queries over large data sets," *Journal of Cloud Computing*, vol. 9, pp. 1–21, 2020.

[20] P. Zhang, J. Fang, C. Yang, C. Huang, T. Tang, and Z. Wang, "Optimizing streaming parallelism on heterogeneous many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1878–1896, 2020.

[21] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 480–491, 2004.

[22] M. Aghbashlo, W. Peng, M. Tabatabaei, S. A. Kalogirou, S. Soltanian, H. Hosseinzadeh-Bandbafha, O. Mahian, and S. S. Lam, "Machine learning technology in biodiesel research: A review," *Progress in Energy and Combustion Science*, vol. 85, p. 100904, 2021.

[23] B. K. Mohanta, D. Jena, U. Satapathy, and S. Patnaik, "Survey on iot security: Challenges and solution using machine learning, artificial intelligence and blockchain technology," *Internet of Things*, vol. 11, p. 100227, 2020.

[24] Z. Chu, J. Yu, and A. Hamdulla, "A novel deep learning method for query task execution time prediction in graph database," *Future Generation Computer Systems*, vol. 112, pp. 534–548, 2020.

[25] R. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1733–1746, 2019.

[26] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?," *Advances in Neural Information Processing Systems*, vol. 35, pp. 507–520, 2022.

[27] L. Yang, S. Liu, S. Tsoka, and L. G. Papageorgiou, "A regression tree approach using mathematical programming," *Expert Systems with Applications*, vol. 78, pp. 347–357, 2017.

[28] A. Paez, F. López, M. Ruiz, and M. Camacho, "Inducing non-orthogonal and non-linear decision boundaries in decision trees via interactive basis functions," *Expert Systems with Applications*, vol. 122, pp. 183–206, 2019.

[29] K. Toutanova and B.-G. Ahn, "Learning non-linear features for machine translation using gradient boosting machines," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 406–411, 2013.

[30] G. Ridgeway, "Generalized boosted models: A guide to the gbm package," *Update*, vol. 1, no. 1, p. 2007, 2007.

[31] K. M. M. Thein, "Apache kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[32] A. V. Tatachar, "Comparative assessment of regression models based on model evaluation metrics," *International Research Journal of Engineering and Technology (IRJET)*, vol. 8, no. 09, pp. 2395–0056, 2021.

[33] D. P. Solomatine and D. L. Shrestha, "Adaboost. rt: a boosting algorithm for regression problems," in *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, vol. 2, pp. 1163–1168, IEEE, 2004.

[34] A. M. Prasad, L. R. Iverson, and A. Liaw, "Newer classification and regression tree techniques: bagging and random forests for ecological prediction," *Ecosystems*, vol. 9, pp. 181–199, 2006.

[35] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[36] J. H. Friedman, "Stochastic gradient boosting," *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[37] E. Scornet, "Trees, forests, and impurity-based variable importance in regression," in *Annales de l'Institut Henri Poincare (B) Probabilites et statistiques*, vol. 59, pp. 21–52, Institut Henri Poincaré, 2023.

[38] A. P. Lind and P. C. Anderson, "Predicting drug activity against cancer cells by random forest models based on minimal genomic information and chemical properties," *PloS one*, vol. 14, no. 7, p. e0219774, 2019.

[39] H. Lee, J. Kim, S. Jung, M. Kim, and S. Kim, "Case dependent feature selection using mean decrease accuracy for convective storm identification," in *2019 International Conference on Fuzzy Theory and Its Applications (iFUZZY)*, pp. 1–4, IEEE, 2019.

[40] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," *Advances in neural information processing systems*, vol. 26, 2013.

[41] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, "Permutation importance: a corrected feature importance measure," *Bioinformatics*, vol. 26, no. 10, pp. 1340–1347, 2010.