A Learning-Based Scheduler for High Volume Processing in Data Warehouse using Graph Neural Networks

Vivek Bengre[1], M.Reza HoseinyFarahabady[2], Mohammad Pivezhandi[1], Albert Y. Zomaya[2], and Ali Jannesari[1]

[1] Iowa State University, Department of Computer Science,
Laboratory for Software Analytics and Pervasive Parallelism (SwAPP)
[2] The University of Sydney, School of Computer Science,
Center for Distributed and High Performance Computing, NSW, Australia
{bvivek2,mpvzhndi,jannesari}@iastate.edu
{reza.hoseiny,albert.zomaya}@sydney.edu.au

Abstract. The process of extracting, transforming, and loading (also known as ETL) of a high volume of data plays an essential role in data integration strategies in data warehouse systems in recent years. In almost all distributed ETL systems currently use in both industrial and academia context, a simple heuristic-based scheduling policy is employed. Such a heuristic policy tries to process a stream of jobs in the best-effort fashion, however, it can result in under-utilization of computing resources in most practical scenarios. On the other hand, such inefficient resource allocation strategy can result in an unwanted increase in the total completion time of data processing jobs. In this paper, we develop an efficient reinforcement learning technique that uses a Graph Neural Network (GNN) model to combine all submitted tasks graphs into a single graph to simplify the representation of the states within the environment and efficiently make a parallel application for processing of the submitted jobs. Besides, to positively augment the embedding features in each leaf node, we pass messages from leaf to root so the nodes can collaboratively represent actions within the environment. The performance results results show up to 15% improvement in job completion time compared to the state-of-the-art machine learning scheduler and up to 20% enhancement compared to a tuned heuristic-based scheduler.

Keywords: Extract Transform Load (ETL) Operations · Scheduling Policy · Data Streaming Processing System · Graph Neural Networks · Job Completion Time · Reinforcement Learning

1 Introduction

The process of extracting, transforming, and loading (also known as ETL) a high volume of data plays an essential role in data integration strategies in data warehouse systems in recent years. A typical ETL process gathers several types

of data from different sources, and then tries to refine and delivers the refined sets to a Data Warehouse (DW) platform (*e.g.*, Amazon Red-shift[1], Azure Data Warehouse Service [2], or Google Big-Query[3]) where the underlying engine allows the end-users to effectively perform the critical business intelligence (BI) activities (such as data predictive analytic). Data processing systems over batch/streaming flows are becoming more and more prominent in the past few years as there is a need to manage apply a set of distributed data mining algorithms over massive data-sets in a petabyte scale.

The range of versatility allows the end-users to submit and run a variety of different algorithms with different load characteristics. In particular, the set of end-users jobs can be scheduled by running a simple scheduling heuristic-based algorithm such as Round Robin (RR), rule based scheduling heuristics, First Come First Serve (FCFS), Shortest Job First (SJF) among others [4, 5]. While on a small scale, the achieved performance of such simple scheduling policy can be considered in an acceptable level, the performance degradation caused by applying such simple policies becomes immediately visible on larger clusters that handle various large workloads on their expensive compute applications. As a result, achieving a near optimal solution that can effectively cope with the challenging issues of dedicating an appropriate number of executors to each job or stage when the arrival rate of jobs or data is unknown in prior is highly desirable.

In most distributed ETL frameworks in data warehouse environments, the set of data processing jobs are broken down into smaller sub-tasks which is known as processing stages. Each of these processing stages can be conceptually linked together to form an abstract processing structure (as a graph) that represents the dependencies between the processing stages. Breaking the submitted processing jobs down into smaller stages/fragments makes them more manageable. Moreover, fragmentation makes it possible to run sub tasks in a concurrent/parallel fashion. In most practical scenarios, such smaller tasks are linked together to form an underlying structure for the application that is usually referred to as a `Directed Acyclic Graph` (DAG). When encoded as DAGs, dedicating jobs to each cluster node is shown to be an NP-hard problem, but we can approximate the solution using graph processing techniques [6]. As such, we use the information present within the job structure to find patterns of efficient execution. Manually traversing all the execution paths to make a decision is not feasible (or extremely slow) for large job sets. Therefore, in this paper, we aim to develop an innovative way to look ahead from the leaf nodes to the root node of the DAG using `Graph Neural Networks` (GNNs) and decide the order of execution.

Original Contribution

In this paper, we develop an efficient embedding plan to reduce the time of convergence and enhances the amount of the reward in each episode of the reinforcement learning (RL) agents. We employ a Double Deep Q-Networks (DDQN)[7] can tune the parameters of the graph neural networks to set the efficient embedding for the DAGs. For any DQN based algorithm to find an efficient policy

(*e.g.*, [8]), it has to explore the state space sufficiently. However, this will make the converging and conforming to a policy take a long time. We use an initial step of the heuristic-based scheduler and reinforcement learning- neural networks agent to assist for efficient policy exploration through the first episode.

Further, we solve the executor limit selection by limiting a stage to one executor and allowing the Agent to select the order. Limiting the number of executors per node allows more executors to be accessible at a given time. We test our model on a simulator built for Apache Spark that also simulates Decima [9]. Our method, Decima, FIFO, and a heuristic-based dynamic partitioning, are compared based on average job completion time, executor usage, and training time.

The main contribution of the current study is summarized as follows.

- We use SageCONV to implement message passing in the reverse direction, which allows us to embed more information in each node for taking actions.
- We make the training process a significant order of magnitude faster by directly representing the Q-values by node feature embedding in the reinforcement learning agent.
- We combine all the DAGs into a single DAG structure to enhance reinforcement learning parallelism and descriptively in-state representation. We train the model by utilizing DDQNs for continuous job arrival.

The rest of this paper is organized as follows. Section 2 highlights the main challenges associated with scheduling of sub-tasks for performing data ETL operations in distributed data processing platforms (such as a data warehouse system). Section 3 presents the details of our proposed scheme. The performance of the proposed solution against famous heuristic-based static and dynamic algorithms is evaluated in Section 4. Finally, Section 5 concludes our work.

2 Problem Statement

The process of data extraction from data sources, transformation, and loading to a central host (commonly known as ETL operations) is among the core strategies and technologies used by enterprises for the data analysis of business information for making business decisions in common Business intelligence (BI) platforms. Business intelligence technologies can handle large amounts of structured/unstructured data to develop and create new strategic business opportunities by easy interpretation of big data sets usually derived from the market in which an enterprise operates (also known as the external data) with data from the internal sources of the business (such as financial and operations data). Such insights can provide enterprises with a competitive market advantage and long-term stability at the broadest level. Common applications of the BI tasks include, but not limited to online analytical processing, data/process/text mining, complex event processing, and predictive/prescriptive analytic. Such applications can empower enterprises to gain insight into new markets or to assess demand and suitability of products and services for different market segments.

Large scale data processing systems can involve a considerable amount of complexity; hence, a significant operational problem can occur when one employs improperly designed data processing systems. Creating an effective scheduling of data processing tasks over limited computing resources across the lifetime of its usage is immensely important in such systems. In particular, an efficient scheduling policy must solve issues such as the decomposition of the original data processing applications to some smaller independent tasks which may be processed in a parallel or distributed manner. Further, thread management, their synchronization and communication can exacerbate the problem as the amount of data becomes larger. Parallel processing of data stream is a very active research topic and there are a myriad of researches that proposed different scheduling strategies to process data streams or real-time data streams [10, 11]. The common requirements for all systems are throughput (efficient utilization of available resources) [12]. The average or p-99 response time becomes the target of some previous researches to address. In the rest of this section, we highlight some main challenges when designing a scheduling policy for a large scale data processing application.

Job scheduling challenge. Scheduling policies can be grouped to two broad categories of either domain-specific [12, 13] or general data processing approaches. The domain-specific policies mostly concentrate on efficient separation of tasks into efficient processing sub-tasks. On the other hand, the general data processing approaches focus on separation of the general jobs into multiple stages and tasks regardless of their intrinsic behavior [11, 14, 15]. The most commonly used scheduler policies in the industrial projects are those that are designed based on simple heuristic-based [4, 5, 16] approaches. Authors in [17–20] propose a control-based approach for guaranteeing the Quality-of-Service (QoS) requirements associated with parallel running queries in distributed stream processing engines and event-driven serverless platforms. Such policies usually simplify the scheduling policies by modeling the task properties based on the embedded features of the jobs. These modeling policies can be improved by considering the dependencies among tasks [21], or making them hybrid with the learning mechanisms [22]. However, it has been proven they are inefficient for complex, and high-frequency job arrival [9]. The current trend is to provide a self-intelligible scheduler to enhance resource allocation through time [15, 23].

Graph structure challenge. Effective handling of task scheduling problems are critically important part of any data processing framework. Because an application can be composed of several partial smaller tasks/operations (also known as the underlying application graph), an optimal scheduler must be optimized accordingly. The goal can be optimizing the utilization of the CPU or memory of the underlying system or to reduce the response time of the tasks (or a combination of both). Having the application graph helps to reduce the model complexity substantially and introduces tools for efficient learning, fast training, and low latency scheduling [24].

Graph Neural Networks. Graph Neural Networks (GNNs) is a deep learning structure that addresses graph-related problems represented via vertices and

an edge regarding dependencies. Graph neural networks have a wide variety of applications in Social network recommendations, node classifications, medicinal drug delivery, and protein-protein interaction. The graph embedding is developed to change the nodes and edges representation of the graphs to preserve information while compressing them down to a manageable size. There are multiple ways in which this embedding can be done, but all the procedures use message passing in some way to include the features in the adjacent nodes. Computing the node embedding is based on the user-specified function, and, similarly, edges can have features of their own, and the embedding for each edge is calculated by considering the connected nodes and the node features themselves [24].

Reinforcement Learning. Machine Learning, in essence, is trying to find patterns in data. Very often, optimal data is required by ML algorithms to make the correct prediction. However, data for the optimal solution does not exist in some instances, such as decision-making environments. The optimum has to be found itself without the correct data. Reinforcement learning algorithms provide a way of interacting with the environment to make decisions and classify a decision as good or bad. Reinforcement learning is always goal-directed and is implemented in an active learning model, i.e., the model learns while interacting with the environment.

Reinforcement learning models that make decisions are called agents. An agent has a state, a policy, a value function, and a model. The actions performed by an agent entirely depend on the state it is in, and this state is not to be mistaken by the environmental state. Environment states are generally not completely visible to the Agent; however, there are cases where the environment state is visible in games like Chess. A policy defines agent behavior and maps from state to action, and it is represented by π in the equations 1 and 2. The value function calculates the expected reward by following π for a state s, and the model predicts what the environment does. The model is never perfect but a good approximation of the environment. For reinforcement learning algorithms, the environment is always considered Markovian, i.e., the current time step represents all the time steps before it.

$$\pi(action|state) = P(action|state) \tag{1}$$

$$v_\pi(s) = E\ [\ G_t \mid S_t\ ,\ A_t\ \rightarrow \pi(s)\ ] \tag{2}$$

In 2 and 3, G_t represents the total expected reward for state S_t and the action A_t as per policy π. G_t can also be expanded as 3 to represent the total expected reward. The discount factor γ, shown in 4, represents the uncertainty with which the reward for the next steps will be computed. The objective of the algorithm is to find the optimal policy π_*.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} ... \tag{3}$$

$$\pi_* = max(\sum_{t>0} \gamma^t R^t) \tag{4}$$

3 Proposed Approach: Design and Analysis

The information present within the structure of the job would help to find efficient patterns of execution. We develop an innovative way to look ahead from the leaf nodes to the root node of the DAG using Graph Neural Networks (GNNs) [6] and decide the order of execution according to an enhanced agglomeration of information. A Double Deep Q-Networks [7] tunes the parameters of the graph neural networks to set efficient embedding for the DAG features. GNNs generalize the conventional deep learning by representing their structure as a set of nodes and edges as their dependencies [25]. The graph neural network can be used to represent the deep neural networks hierarchically to reduce the complexity of training by creating replicated kernels [26–29]. Besides, the stages are limited to one executor, and the Agent decides to dedicate free executors after resources are allocated. To deal with time-consuming convergence in search for an efficient policy on the proposed approach in [8], we propose a hybrid heuristic-based scheduler to assist by executing for the first few episodes. Along with creating a large DAG structure, we also utilize a state representation that helps us to parallelize the training and inference processes.

3.1 Preliminaries

Apache Spark is one of the most widely used open-source computing engines. Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object, that is in the main program. To run a Spark engine on a cluster, the `SparkContext` object needs to connect to the cluster manager. `SparkContext` object can connect to YARN, Mesos, Kubernetes or Spark's default Standalone manager. Once the nodes are connected to the cluster manager, Spark engine acquires executors on the worker nodes in the cluster, which are processes that run computations and store data for the application. Then, the `SparkContext` object sends the submitted tasks to the executors to be executed.

Spark engine provides a rich selection of APIs and libraries that support Extract Transform Load (ETL) operations, graph computations, streaming jobs, real-time query processing, and Machine Learning capabilities. The model proposed in this paper would be tested based on an accurate simulator built for Apache Spark [9]. The comparison metrics would be average job completion time, executor usage, and training and inference time. Our model would be compared with various heuristic-based schedulers, including First in First out (FIFO), dynamic partitioning algorithms, and the state-of-the-art reinforcement learning-based scheduler named `Decima`. The proposed workflow in this paper is represented in Fig. 1.

3.2 State Representation

The environment state at any given point contains all the job DAGs that have not been executed. Each job DAG is a sparse matrix of edges and vertices, along

Entity	Symbol
Discount factor	γ
Action	A_t
Policy function	π
State	S_t

Table 1: Notation used in the paper

with a matrix of features for each node. An environment state can be represented as a collection of sparse matrices with their corresponding feature matrices. The proposed solution in this paper uses something similar to describe the environment state, i.e., it uses one large graph containing all the DAGs present i.e., DAGs that have not been fully executed. A flag is changed to indicate it has completed its execution. The "soft" delete is done to keep the node numbering and positions correct. In the Fig 3.2, the node numbering for some nodes has been repeated, and these repetitions are representative of the new job arrivals. To identify nodes internally, they've been numbered from 0 to node count where node count is the total number of nodes in the graph. Having this structure allows for adding new jobs quickly by appending them to the existing list. The job features keep changing as new jobs arrive and the cluster state changes. So, it's better to compute the feature matrix just before training or inference. The process of feature calculation is also not expensive as most frameworks provide this information about the node. This aggregated state representation also allows an efficient way to parallelize and compute the embedding in the next step. The final step is to reverse the directions of all the edges in the graph, this is required for the leaf embedding to have influence from the higher nodes.

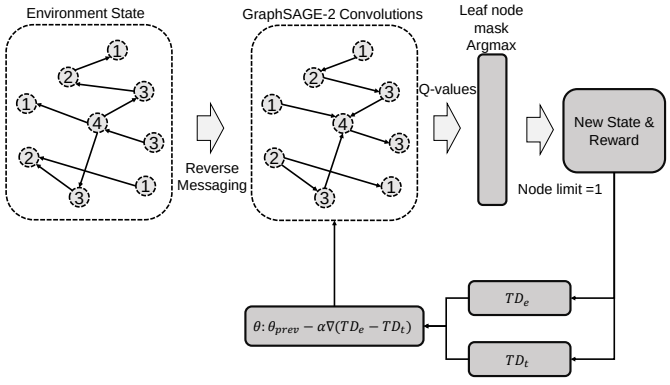

Fig. 1: The workflow and the structure of the proposed solution.

3.3 Graph Embedding

A graph neural network is used to embed each node information as logits. In the case of our solution, the resulting logits are just one number which represents the Q value for selecting a node as its action. The graph neural network takes job dags and the features of each node as input and outputs a Q value for each node. In the graph neural network, we adopted three GraphSAGE [30] layer. In first two layers, the number of input and out features are five while last layer takes five features from the output of second layer as input but it outputs only one feature which is the Q value or logits for each node.

Since the Message Passing path is reversed, the leaf node values calculated will have influence from nodes that are a few generations (in terms of dependency) above it. So, the leaf node value will represent the "path" from the leaf to some parent node. For any node in the graph the embedding is calculated as follows.

$$x^l_{parent(u)} = agg(\ \{\ x_v^{l-1}\ \forall\ v\ \epsilon\ parent(u)\ \}\ ) \tag{5}$$

$$x^l_u = \sigma(\ W\ .\ concat(x_u^{l-1}, x^l_{parent(u)})\ ) \tag{6}$$

The N or the Neighborhood of a given node automatically changes to the parents/dependants of the node. One round of message passing will not be enough for the leaf nodes to have enough influence from the nodes that are higher up. So, to have a reasonable influence, three rounds of message passing is done. This assures an embedding that will take into account the neighbourhood that spans reasonably away from the leaf nodes. The next step is to train the embedding to give accurate/efficient Q-values per node.

4 Performance Evaluation Results

The proposed approach is based on the Decima spark simulator [9]. We compared the results with FIFO as spark default scheduling, dynamic partitioning scheduler, and Decima. The executor usages, average job completion time, and cumulative distribution of the rewards are three significant evaluated criteria. The jobs are generated randomly based on the TPC-H dataset [31], and the rewards may be increased based on extending the generated job set. The proposed solution includes the same randomness of input jobs, and the evaluation is based on the average ratio for the improvement over multiple runs.

Instead of focusing on matrix factorization, which is a common embedding technique in GCN, we use an inductive method based on node features in GraphSAGE [30] to learn the embedding features that would generalize to unseen nodes. Our model is based on an aggregation of feature information based on the neighboring nodes, and the back-propagation by stochastic gradient descent is used to train the parameters. The symmetric aggregator function makes the model trainable by ordering the unordered set of vectors as the neighbors of each

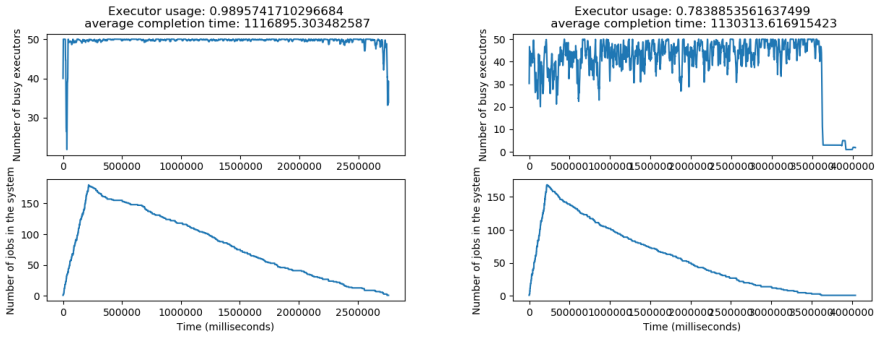

Fig. 2: Performance Evaluation of Decima Executor usage verus Dynamic Scheduling

Table 2: Parameters for different training stages based on pooling and mean aggregator stages.

Parameter	Pooling	Mean	
Stage	-	1	2
Burning	1000	1000	1000
Learning Rate	0.001	0.001	0.001
Episodes	0.001	15	30
Gamma	0.9	0.9	0.9
Assist	90%	100%	0%
Random Exploration	10%	0%	100%
Exploration Decay	0.9999	0.9998	0.9998

node. We considered two different aggregator functions, mean and pooling, to train the models. Our experimentation in Fig. 3 shows that the pool aggregator requires more training episodes, and the loss value converges considerably slower than the mean aggregator. However, the convergence policy is comparable in terms of the efficiency of scheduling. The parameters are initialized as provided in Table 2.

5 Conclusion

Graphinator successfully reduces the job completion time in high-frequency job arrival cases. Our results show that having a graph neural network computing the Q-value helps execute jobs much more efficiently. Irrespective of the number of parallel nodes assigned, this work also shows that with the assistance of optimized scheduling algorithms, the training time for a model can drastically be reduced. We also show that assigning one executor per stage in a job DAG works well for high-load environments. However, we also observed that the overall response time (makespan) of the jobs is the limiting factor of assigning one

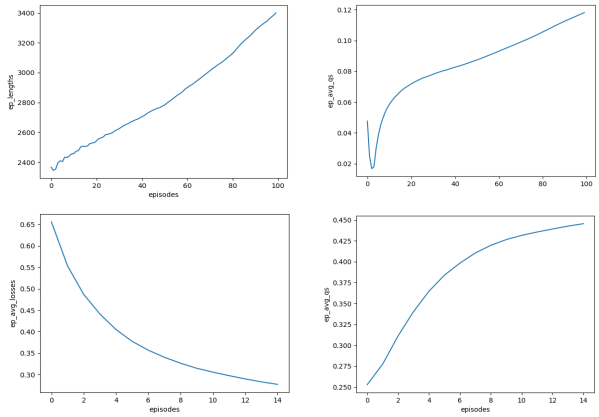

Fig. 3: GraphSAGE with pool aggregator converges fairy fast in the beginning episodes then stabilizes. The left side figure shows the average losses plot, the right side figures represent the average QS value. Top images represent training via pooling aggregator and the down images are stage 1 and stage 2 for mean aggregator.

node per stage. This limitation can be solved by manually tuning the algorithm for lower loads and increasing the maximum number of executors per stage. The ability to learn of our model helps to efficiently enhance its performance for an extended duration of time with more randomized real-life cluster loads. As future work, the proposed method can be continued to optimize hardware requirements with limited memory, CPU, and storage.

Acknowledgment

We thank the Research IT team *(ResearchIT – RIT)* of Iowa State University for their continuous support in providing access to HPC clusters for conducting the experiments of this research project. Prof. Albert Y. Zomaya acknowledges the support of Australian Research Council Discovery scheme (DP190103710). Dr. MohammadReza HoseinyFarahabady acknowledge the continued support and patronage of *The Center for Distributed and High Performance Computing* in *The University of Sydney, NSW, Australia* for giving access to advanced high-performance computing platforms and industry's leading cloud facilities, machine learning (ML) and analytic infrastructure, the digital IT services and other necessary tools.

References

1. Amazon redshift: cloud data warehouse. https://aws.amazon.com/redshift/. Accessed: 2021-10-25.
2. Azure data warehousing architectures. https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/data-warehousing. Accessed: 2021-10-25.
3. Bigquery: Cloud data warehouse. https://cloud.google.com/bigquery. Accessed: 2021-10-25.
4. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.
5. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. page 2, 2012.
6. Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.
7. Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
8. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
9. Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.
10. Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th Intl. Conference on Distributed Computing Systems (ICDCS)*, pages 122–132. IEEE, 2019.
11. Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.
12. Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. Dl2: A deep learning-driven scheduler for deep learning clusters. *arXiv preprint arXiv:1909.06040*, 2019.
13. Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.
14. Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Intl. Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.
15. Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

16. Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.
17. Mohammad Reza Hoseiny Farahabady, Albert Y. Zomaya, and Zahir Tari. Qos- and contention- aware resource provisioning in a stream processing engine. In *Intl. Conf. on Cluster Computing*, pages 137–146, 2017.
18. Yidan Wang, Zahir Tari, M. Reza HoseinyFarahabady, and Albert Y. Zomaya. Qos-aware resource allocation for stream processing engines using priority channels. In *Intl. Symposium on Network Computing and Applications (NCA)*, pages 1–9, 2017.
19. M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Trans. on Parallel and Distributed Systems*, 29(7):1442–1455, 2018.
20. Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, Albert Y. Zomaya, and Raja Jurdak. Dynamic control of cpu usage in a lambda platform. In *Intl. Conf. on Cluster Computing (CLUSTER)*, pages 234–244, 2018.
21. Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016.
22. Neetesh Kumar and Deo Prakash Vidyarthi. A novel hybrid pso–ga meta-heuristic for scheduling of dag with communication on multiprocessor systems. *Engineering with Computers*, 32(1):35–47, 2016.
23. Bingqian Du, Chuan Wu, and Zhiyi Huang. Learning resource allocation and pricing for cloud profit maximization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7570–7577, 2019.
24. Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.
25. Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.
26. Sixing Yu, Phuong Nguyen, Waqwoya Abebe, Ali Anwar, and Ali Jannesari. Spatl: Salient parameter aggregation and transfer learning for heterogeneous clients in federated learning, 2021.
27. Sixing Yu, Arya Mazaheri, and Ali Jannesari. Auto graph encoder-decoder for neural network pruning. In *Proc. of IEEE/CVF Intl. Conf. on Computer Vision (ICCV)*, pages 6362–6372, October 2021.
28. Sixing Yu, Arya Mazaheri, and Ali Jannesari. Auto graph encoder-decoder for model compression and network acceleration. *arXiv preprint arXiv:2011.12641*, 2020.
29. Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.
30. William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st Intl. Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.
31. Tpc-h vesion 2 and version 3.