DBMLSched: Scheduling In-database Machine Learning Jobs [Extended Abstract]

Yunjia Zheng, Yuxuan Tian, Joseph V. D'Silva and Bettina Kemme

McGill University, Montreal, Quebec, Canada

Abstract

Given the large amount of data that resides in relational database management systems (DBMS) and the fact that the DBMS often run on powerful servers, there exist considerable efforts for integrating machine learning (ML) support into the DBMS. This is particularly attractive in the explorative phase of data analytics that experiments with different ML algorithms on various subsets of the data. However, collocating ML and query processing on the same machine requires a scheduling mechanism that considers resource consumption, reasonable response times for interactive learning and the option to exploit both CPU and GPU. We propose DBMLSched, a scheduling mechanism that performs ML runtime prediction whenever possible and carefully monitors jobs and resources in the system, dynamically allocating ML jobs to their optimal device and pausing jobs if any performance interference occurs while at the same time avoiding starvation. Our preliminary results show a considerably improved response time compared to executing ML jobs in their order of arrival with less negative impact on concurrent queries.

Keywords

Database Management System, Machine Learning, Scheduling

1. Introduction

Many popular machine learning frameworks including PyTorch, TensorFlow, Pandas and Spark load data into their own execution environments, often on the user's own limited machine [1, 2, 3, 4]. If the data resides in a relational DBMS this involves expensive data transfer. Thus, there is a rising trend to integrate parts of an ML workflow inside the database engine to reap the benefits of a *near-data* approach [5, 6, 7, 8, 9, 10]. However, ML jobs deviate significantly from traditional database workloads in their computational and resource requirements making it necessary to rethink how to perform task scheduling.

In this paper, we focus on the *exploratory phase* of data science often performed in an interactive mode, and that experiments with a variety of ML learning jobs, exploring various subsets of the available data and smaller data sets, as a precursor to deciding which data sets and attributes are relevant to the problem. Thus, in our system, we expect many different variations of ML algorithms that run fairly short. We believe that it is beneficial to execute such jobs in the DBMS, as to exploit the DBMS query capacity to select data subsets and to avoid data transfer. These ML jobs must then co-exist with traditional database queries.

Both ML and query scheduling has received wide attention. However, ML scheduling often focuses on long-running deep-learning jobs that are distributed on GPU/CPU clusters [11, 12, 13, 14], while query scheduling approaches do not take concurrent ML jobs into consideration [15, 16].

To enable effective scheduling of explorative ML jobs concurrently with DB queries, a scheduler needs to have several capacities. First, it ideally can predict the runtime of random ML jobs on both CPU and GPU [17]. As a very general solution that is not restricted to specific algorithms or systems' traits, we suggest to exploit the iterative behavior of ML algorithms and execute the first few iterations of the learning loop as an indicator of how long the overall job will take. Given that information we can use a shortest-based first scheduling variation for best average response times. Second, long-running jobs should not delay short-running jobs but also not starve. For that, we propose to split them into smaller jobs and interleave them with other execution. Third, for some jobs runtime prediction might not be possible, therefore we also split them into smaller tasks should they run too long. Fourth, concurrently running jobs might interfere with each other. With many ad-hoc, short-running ML algorithms using different ML libraries and packages, a deep analysis of the programs and detailed distribution of tasks to compute resources for concurrency is not feasible. Instead, we propose a reactive scheduler that reduces concurrency whenever interference is detected. Finally, ML jobs can also interfere with concurrent DB queries. The scheduler must ensure that concurrent query processing is not significantly impacted by a learning task,

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Applied AI for Database Systems and Applications (AIDB'23), August 28 - September 1, 2023, Vancouver, Canada

yunjia.zheng@mail.mcgill.ca (Y. Zheng); yuxuan.tian@mail.mcgill.ca (Y. Tian); joseph.dsilva@mcgill.ca (J. V. D'Silva); bettina.kemme@mcgill.ca (B. Kemme)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3462/AIDB3.pdf

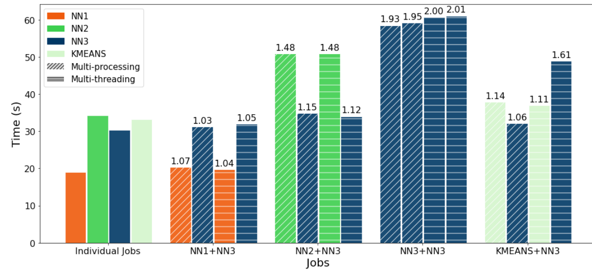

Figure 1: Execution time of concurrent ML jobs

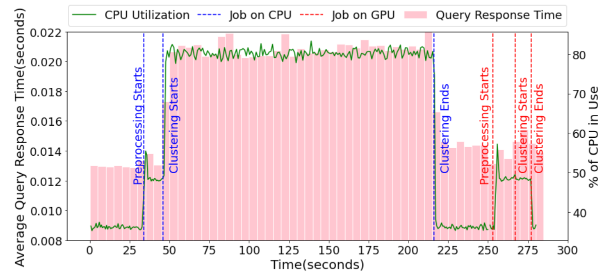

Figure 2: SQL Queries along k-means clustering

in particular when it is executing on the CPU. Again, we propose the scheduler to be reactive and pause learning whenever database queries experience too much delay.

We have integrated DBMLSched, an initial implementation of these ideas, into AIDA [8], a Python-based in-database analytic system residing inside a database engine. In our preliminary evaluation, DBMLSched was able to provide response times that were up to 2.9x faster than if the jobs are executed in arrival order on the GPU while impacting SQL queries to an acceptable degree.

2. Characteristics and Challenges

In the interactive environment that we envision, relational queries and ML jobs can be submitted simultaneously by different clients. In this section, we investigate several characteristics and challenges of executing them concurrently.

Inter-job Interference. We first have a look at how concurrent ML jobs can impact each other. Our experiments use three neural networks NN1-NN3 with different input sizes and iterations, along with a K-Means algorithm. The ideal device for NN1, NN2 and K-Means is the GPU due to the large amount of training data and complex computations. NN3 is a very small job and runs faster on the CPU.

Baseline Experiment. The first 4 columns in Figure 1 show the execution times for all four jobs when executed individually on their preferred device. NN1 and K-means use the GPU fully while NN2 only uses 78%. All three require 10% CPU utilization. NN3 uses only 50% of the CPU (half of the machine's cores).

Concurrent Execution. We always run only one GPU-based job given the large GPU utilization of each, and then run NN3 concurrently on the CPU. In theory enough CPU capacity is available to not incur any interference. We consider two scenarios, one where the ML jobs run within the same process using different threads, and the other where the ML jobs run in different processes. ML execution frameworks outside a DBMS will likely use a multi-process (or even multi-container) approach for isolation purposes. However, DBMS often run in a single process; thus in-database analytics will likely be based on multi-threading for concurrency. The right columns of Figure 1 show the execution times for the various combinations. The numbers above the bars indicate how much slower the job runs compared to when it is run in isolation.

We can see that the different combinations lead to quite different interference, with no interference between NN1 and NN3 as we originally anticipated, with significant interference between NN2 and NN3 for both multi-threading and multi-processing, and with K-means and NN3 interfering little with multi-processing but significantly with multi-threading. Running two NN3 jobs on the CPU also causes interference. The underlying reasons for these unexpected interferences can be related to various causes such as bad concurrency within Python, unexpected interactions between CPU and GPU, for instance for NN2, or suboptimal assignment of cores to different threads/processes.

Summary. While resource utilization can guide scheduling decisions, interferences can occur unexpectedly. A system that needs to schedule unknown ML programs and wants to support arbitrary ML libraries whose internals are not known, cannot simply concurrently run multiple jobs even if sufficient resources appear to be available. Hence, monitoring the system state and adjusting scheduling dynamically will offer better flexibility.

Interference of ML jobs and SQL queries To illustrate how ML jobs affect concurrently executing SQL queries, we have the DBMS host the data set of the TPC-H benchmark and repetitively submit the medium-complex query 17 of the benchmark. In parallel, we submit twice a data science workflow also on the TPC-H data set. The pre-processing phase, among other things, submits SQL queries to retrieve the relevant learning data; learning uses K-means clustering. Pre-processing always executes on the CPU while the clustering job in the first submission executes on the CPU, and in the second submission on the GPU.

Figure 2 shows query response time (pink bars), and CPU utilization over time. The vertical lines mark when the pre-processing and clustering start and end. Response time for query 17 is 13 ms when it runs alone, increases slightly when the workflow starts pre-processing (concurrent SQL queries), and then increases heavily when learning starts on the CPU. In contrast, when we execute clustering on the GPU, there is little impact on query response time. The result is directly related to the CPU utilization caused by the different phases of the data science workload, that reaches 80% only when learning occurs on the CPU.

For interactive environments, query response times are an important metric. For short queries, this might not be an issue but for queries where the base response time is already close to a perceptible threshold, the impact due to long-running ML jobs may be not acceptable, and it might be desirable to hold back the execution of ML jobs. However, real-time monitoring of the query response times requires tight integration with the database execution engine and will have to be custom designed for each database engine. Thus, we propose a more modular but still effective approach where the ML job scheduler dynamically monitors the CPU utilization of the system in order to make its scheduling decisions.

Predicting Execution Times of ML Jobs

While there is some work to predict the execution times of specific ML algorithms [18, 12], working with a wide variety of ad-hoc ML algorithms makes applying them challenging. However, it has already been observed that individual iterations of many popular learning algorithms often take similar amount of time. Indeed, we trained 6 models with different parameters ranging from neural networks to clustering, both on CPU and GPU with increasing number of iterations. The actual execution times depend heavily on the ML algorithm, the input data sizes, and whether execution is on CPU or GPU, but all show linear behavior in regard to the number of iterations.

ML programmers might indicate a certain condition that needs to be fulfilled to stop training (e.g., prediction error under a threshold). The number of iterations is thus determined during run-time. However, explicitly setting such termination conditions might not always be appropriate. Given the variety of algorithms, data scientists might not know in advance what a good termination condition is. Further, while exploring, they likely want to keep the execution times relatively short, in order to produce quick assessments for various algorithms. Thus, we believe that while not all, many ML programs will actually have a predefined number of iterations. We refer to them as **explicit jobs**. For these explicit jobs, once we know how long one iteration takes we can make a good estimation of the overall execution time. In contrast, jobs that work with termination conditions or for jobs that do not follow the paradigm that all iterations have the same length (that might be the case, e.g., with decision trees), overall runtime is difficult to predict. We refer to them as **implicit jobs**.

3. DBMLSched

We propose DBMLSched as a scheduling mechanism that controls the execution of ML jobs within a DBMS. As ML jobs arrive DBMLSched assigns them to either GPU or CPU. As the characteristics of the jobs are not known in advance and the DBMS might at the same time experience various loads in regard to database queries DBMLSched has an observer component that monitors the CPU/GPU utilization overall and of the individual jobs. This information is used by the scheduler to decide where to execute a job and also to halt jobs if there is interference.

Integration into AIDA. AIDA [8] is a data science platform that facilitates the execution of analytical tasks within the database system. AIDA clients use a standardized set of Python APIs and interactive programming environments such as Jupyter to perform linear and relational algebra transformations or write complete ML functions. All calls/functions are shipped to the AIDA server that resides inside the embedded Python interpreter of the DBMS. It executes relational algebra operations using the DBMS query engine while relational algebra operations and ML functions are executed within the Python environment using existing ML libraries such as NumPy and PyTorch. Data remains within the DBMS space unless explicitly requested by the client. We have integrated DBMLSched into the AIDA server. We have also slightly adjusted the Pytorch-based API provided by AIDA in order to facilitate easy extraction of the iterative component of an ML job. On the server side, these ML jobs are then intercepted and controlled by DBMLSched.

Scheduler Overview. The overall idea of DBMSched is (1) to estimate the run-time of newly incoming jobs for both CPU and GPU, to (2) schedule them so that the overall service time is short and to (3) readjust when new jobs arrive or interference is observed. Given the complex interference we have observed in regard to concurrent ML jobs our current solution lets at most one job run on each CPU and GPU. We first outline our proposal assuming only explicit jobs and then extend the solution to also handle implicit jobs.

Runtime estimation When a new job arrives, DBMSched first checks whether it runs faster on CPU or GPU and estimates runtime. To do so, it pauses for each device the current job and executes the new job for a specific time. The current job is paused at the end of its current iteration because recent work [19] has shown that

memory usage is the lowest at the end of an iteration, leading to less overhead when switching jobs. The length of this *estimation phase* is set dynamically such that if the new job is significantly shorter than the current job, it might actually finish during estimation. Otherwise, DBMLSched checks how many iterations have been completed during that time and confirms that they have the same length. From there it determines device preference and run-time. Note that if the individual iterations of the job do not have the same execution time or the job does not at least finish two iterations during the estimation phase, then we cannot make any time prediction, and the job becomes an implicit job.

Scheduling jobs The service time for a job is the sum of its execution time and all the execution times of jobs that execute before it in a given device. DBMLSched keeps two waiting queues, one for the GPU and one for the CPU. Whenever a new explicit job arrives with estimated execution times on GPU and CPU, DBMSched reorders all jobs in both the GPU and CPU waiting lists according to the shortest remaining service time first (SRTF), a preemptive version of the shortest job, as it minimizes the average amount of time each job has to wait until its execution is complete when new jobs arrive on a continuous basis. Thus, we not only consider whether a job runs faster on the GPU or the CPU but also consider its waiting time. Thus, a job might run faster on the GPU than on the CPU but could be still scheduled on the CPU because it would have to wait longer to even start on the GPU.

Whenever a job completes on the GPU resp. CPU, the scheduler takes the first job from the corresponding queue (if there is any) and usually executes it until completion. A new job, however, might preempt the currently running job as described before.

Implicit jobs For implicit jobs, we can determine during the estimation phase whether they run better on the CPU or the GPU but we cannot estimate their run-time. Therefore, we split their execution into several tasks, each of which runs for a limited time, to avoid that they delay short-running jobs for too long. We guarantee that whenever they are scheduled at least one iteration is executed in order to guarantee progress. In order to ensure that our system is starvation free, we treat explicit jobs that have been staying in the system for a long time as implicit jobs.

We maintain separate queues for implicit jobs and schedule jobs for one device in a round-robin fashion, i.e., one explicit job, which will fully complete execution, and then one implicit job, which will execute for a quota of time and go to the end of the implicit queue. Service time calculations are accordingly adjusted considering in each round the time quota an implicit job will get.

Observer Overview. The DBMLSched observer monitors CPU and GPU utilization and responds to high utilization rates, which could be an indicator of resource contention.

The GPU utilization is recorded during the estimation phase since we do not run any other ML job on the GPU, while the difference between the CPU utilization during and before estimation is used as an estimation of the actual CPU utilization of the job, in case CPU is also occupied by queries or pre-processing of other ML jobs. At any given time, the observer keeps track of the estimated CPU utilization of the jobs that are currently running on CPU and GPU.

Interference with queries If the actual CPU utilization is much higher and above a certain threshold, then this indicates that there are concurrent SQL queries and / or pre-processing tasks which might be impacted by the ML job running on the CPU. Thus, the job on the CPU is paused. Once the CPU utilization falls again below this threshold, ML job execution on the CPU resumes.

Interference with ML jobs As observed in Section 2, ML jobs running on the GPU might be impacted by jobs running on the CPU even if they only use a relatively small amount of CPU. In order to detect such interference, we observe GPU utilization. If it is significantly lower than the estimated GPU utilization and a job is running on the CPU, we will pause the CPU job until the job running on the GPU has been completed. Once the GPU has finished, rescheduling happens as described above. With this, jobs that were assigned during the last scheduling phase to the CPU might now be reassigned to the GPU. We prioritize the GPU because we assume that most jobs run significantly faster on the GPU and the CPU might mainly be a backup resource. Should there be many jobs that have true CPU affinity, other prioritization can be used.

4. Preliminary Experiments

We first look at how our scheduler can address interference with SQL queries. Figure 3 shows an execution over time where at the beginning a (blue) ML job is executed on the CPU and then a (red) ML job is scheduled on the GPU. Then first a lightweight SQL workload is activated causing a CPU utilization of roughly 20% if run alone in the system and then a heavier SQL workload with complex queries requiring roughly 40% of the CPU. If there is no observer (Fig. 3(a)) query response times are slightly higher for the light workload than if no ML job is run on the CPU (orange vs. add-on blue bars), while the more complex queries are heavily impacted by the ML job running on the CPU, having nearly 50% larger response times. In contrast, the observer (Fig. 3(b)) detects that the CPU threshold is passed and the ML job on the CPU is stopped, which prevents the severe delay.

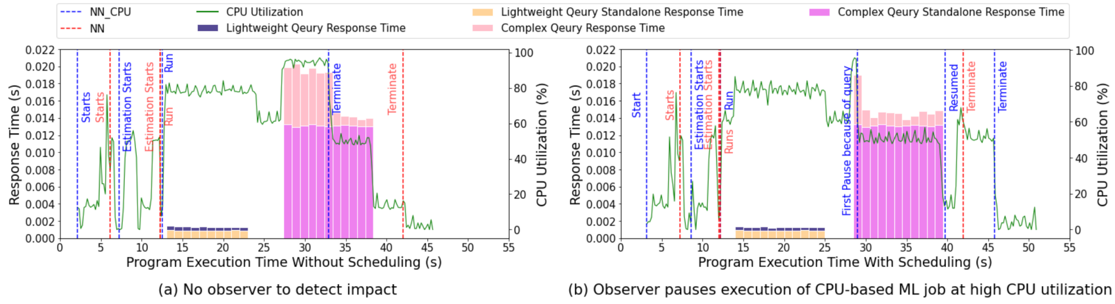

Figure 3: SQL queries concurrent to ML jobs on GPU and CPU

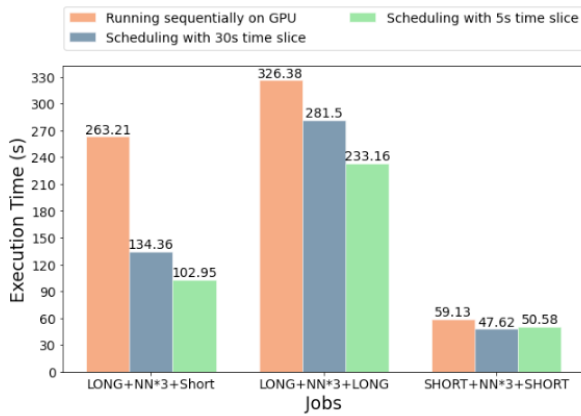

Figure 4: Average service times using different quotas

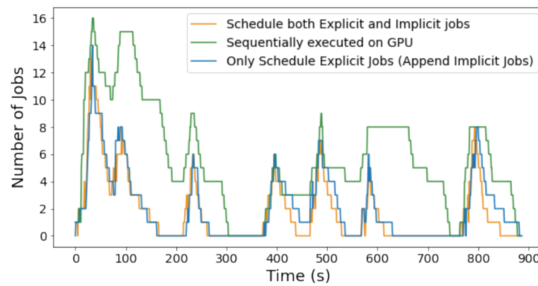

Figure 5: The number of jobs in the system

In the next experiment, we look at how the scheduling policy takes effect on a hybrid workload of implicit and explicit jobs, and also how the choice of quota affects overall behavior. Our workload consists of 5 machine learning jobs: two implicit jobs LONG/SHORT submitted first and last with three identical explicit neural network (NN) jobs in between, with a 3-second submission interval. Figure 4 shows that DBMLSched always performs better than sequential execution and short quotas for implicit jobs behave better than longer ones in 2 of the three situations. The shorter quotas allow the shorter jobs in the implicit and the explicit queues to be less delayed by the LONG job. Of course, one has to be careful not to set the quota too short as this could then delay the long job in case short jobs come into the system continuously. The values chosen for the time quota of implicit and long jobs clearly influence the prioritization of long/implicit jobs vs. short explicit jobs and are something that could be adjusted depending on the typical workload that might be submitted to the system and user expectations.

Lastly, we experimented during a 15-minute period to test the stable behavior of our scheduling system. The workload here consisted of a set of neural networks (both implicit and explicit), K-Means jobs and regression jobs of various lengths. Overall, service times using DBMLSched were 2.9x faster than when jobs are executed on the GPU sequentially in the order of arrival. Figure 5 shows the number of jobs that were in the system at any time during the experiment. We can see that relatively early in the experiment many jobs were submitted and our scheduler was able to work them off much quicker than sequential scheduling, keeping the waiting queues much shorter throughout the experiment. The reason is that our scheduler executes the shorter K-Means jobs first and also executes some jobs on the CPU, relieving the pressure on the GPU. Also as the short implicit jobs are executed in a round-robin fashion with the explicit jobs, they complete quickly.

5. Conclusion and Future Work

DBMSched is a resource-aware scheduler that can handle arbitrary jobs and observes their behavior to perform scheduling decisions. In future work we would like to extend it to also consider memory aspects. Better control over which CPU cores are used by which tasks might also help in avoiding interference.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS), 2019, pp. 8024–8035.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, Tensorflow: Large-scale machine learning on heterogeneous distributed systems, CoRR abs/1603.04467 (2016).

[3] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project, CoRR abs/1309.0238 (2013).

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (2020) 357–362.

[5] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, A. Kumar, The madlib analytics library or MAD skills, the SQL, VLDB Endow. 5 (2012) 1700–1711.

[6] X. Li, B. Cui, Y. Chen, W. Wu, C. Zhang, Mlog: Towards declarative in-database machine learning, VLDB Endow. 10 (2017) 1933–1936.

[7] M. E. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, S. Günnemann, In-database machine learning with SQL on GPUs, in: International Conference on Scientific and Statistical Database Management (SSDBM), ACM, 2021, pp. 25–36.

[8] J. V. D'silva, F. De Moor, B. Kemme, AIDA - abstraction for advanced in-database analytics, VLDB Endow. 11 (2018) 1400–1413.

[9] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, V. Srinivasan, Amazon Redshift and the case for simpler data warehouses, in: ACM International Conference on Management of Data (SIGMOD 2015), 2015, pp. 1917–1923.

[10] A. Yakovlev, H. F. Moghadam, A. Moharrer, J. Cai, N. Chavoshi, V. Varadarajan, S. R. Agrawal, T. Karnagel, S. Idicula, S. Jinturkar, N. Agarwal, Oracle automl: A fast and predictive automl pipeline, VLDB Endow. 13 (2020) 3166–3180.

[11] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Guo, Optimus: An efficient dynamic resource scheduler for deep learning clusters, in: EuroSys Conf., 2018.

[12] H. Zhang, L. Stafman, A. Or, M. J. Freedman, SLAQ: Quality-driven scheduling for distributed machine learning, in: Symp. on Cloud Computing (SoCC), 2017, p. 390–404.

[13] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, M. Zaharia, Heterogeneity-aware cluster scheduling policies for deep learning workloads, in: USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2020.

[14] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, et al., Dynamic scaling on GPU clusters for deep learning, in: USENIX Symp. on Operating Systems Design and Impl. (OSDI), 2020.

[15] B. Wagner, A. Kohn, T. Neumann, Self-tuning query scheduling for analytical workloads, in: ACM Int. Conf. on Management of Data (SIGMOD), 2021, pp. 1879–1891.

[16] I. Sabek, T. S. Ukyab, T. Kraska, LSched: A workload-aware learned query scheduler for analytical database systems, in: ACM Int. Conf. on Management of Data (SIGMOD), 2022.

[17] T. Lim, W. Loh, Y. Shih, A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms, Mach. Learn. 40 (2000) 203–228.

[18] M. Lattuada, E. Gianniti, D. Ardagna, L. Zhang, Performance prediction of deep learning applications training in GPU as a service systems, Clust. Comput. 25 (2022) 1279–1302.

[19] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, et al., Gandiva: Introspective cluster scheduling for deep learning, in: USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2018, pp. 595–610.