2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00166

Rotary: A Resource Arbitration Framework for Progressive Iterative Analytics

Rui Liu
University of Chicago
ruiliu@cs.uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Michael J. Franklin
University of Chicago
mjfranklin@uchicago.edu

Sanjay Krishnan
University of Chicago
skr@cs.uchicago.edu

***Abstract*—Increasingly modern computing applications employ progressive iterative analytics, as best exemplified by two prevalent cases, approximate query processing (AQP) and deep learning training (DLT). In comparison to classic computing applications that only return the results after processing all the input data, progressive iterative analytics keep providing approximate or partial results to users by performing computations on a subset of the entire dataset until either the users are satisfied with the results, or the predefined completion criteria are achieved. Typically, progressive iterative analytic jobs have various completion criteria, produce diminishing returns, and process data at different rates, which necessitates a novel *resource arbitration* that can continuously prioritize the progressive iterative analytic jobs and determine if/when to reallocate and preempt the resources. We propose and design a resource arbitration framework, Rotary, and implement two resource arbitration systems, Rotary-AQP and Rotary-DLT, for approximate query processing and deep learning training. We build a TPC-H based AQP workload and a survey-based DLT workload to evaluate the two systems, respectively. The evaluation results demonstrate that Rotary-AQP and Rotary-DLT outperform the state-of-the-art systems and confirm the generality and practicality of the proposed resource arbitration framework.**

***Index Terms*—scheduling, deep learning, approximate query processing, progressive analytics, model training, query execution**

I. Introduction

A growing concern for organizations is high resource usage in data analytics [1]–[4]. The concerns have become particularly acute over the last two years where a confluence of factors, including supply chain shortages and a waning Moore's law, have discouraged organizations from simply scaling out to cope with the ever-increasing analytics demands. In this resource-limited world, every organization needs to determine how to partition and share computing infrastructure to adequately support all of its analytics users [5]–[9]. Despite this importance, existing scheduling and resource allocation approaches do not adequately support modern data analytics workloads.

From aggregate statistics to machine learning, modern data analytic jobs embrace approximation and uncertainty. They are often progressive, where an iterative loop repeatedly refines an answer until the desired completion criterion is met. In this setting, job completion is a matter of user opinion, where a user-defined rule has to be used to terminate the job when the answer is deemed sufficiently accurate or unchanged.

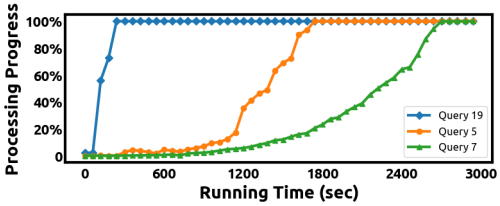
A traditional job scheduler places an immense level of trust in a user's ability to decide when to terminate these progressive iterative analytic jobs appropriately, which may bring disastrous consequences. For example, consider a user training a convolutional neural network for a fixed time of 500 epochs. Suppose the model actually converges in accuracy after only 100 epochs; then 80% of this model's training time is a wasteful block on system resources. Similarly, the same problem can occur in approximate query processing systems. Suppose a user has been given a time budget of three minutes to complete a reporting query over a data warehouse, but the query result is precise enough for the application after one minute. For these progressive iterative analytic jobs, overly ambitious completion criteria can block key resources from other users for an extended amount of time.

An ideal scheduler for progressive iterative analytic jobs needs introspection into the convergence progress of each job in the queue to be able to detect and preempt such anomalies adaptively. These decisions need to consider a job's prioritization, specified completion criteria, the available resources, and other jobs waiting for the resources — a problem we call *resource arbitration*, which is a novel adaptive and completion criteria-aware scheduling paradigm. We identify two widely-used applications that fit this resource arbitration paradigm: approximate query processing (AQP) and deep learning training (DLT). In AQP, one executes queries on a subset of the overall dataset or a data stream to return an approximate answer within a user-specified error. In DLT, one updates the parameters of the neural network-based model with a variant of gradient descent repeatedly until the desired objective (e.g., accuracy or convergence) is reached. In both of these scenarios, one needs a resource arbitration system that *can pause a running job at the risk of dequeuing it in a partially complete state, in favor of jobs that could better use the same resources, especially in resource-constrained environments*.

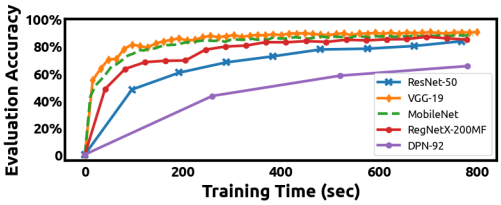
This strategy is only useful in a setting where an intermediate result has significant utility to a user, as is the case in progressive iterative analytic jobs. We plot the progress curve of sample AQP and DLT jobs in Fig. 1 to demonstrate this trait. As the job progresses (and consumes more resources), the incremental utility of each additional processing-second spent decreases. These diminishing returns have to be factored into the scheduling algorithm, especially if there is another job in the queue that could make more significant progress if allocated those same resources.

Motivated by the unique traits of progressive iterative analytic jobs, we propose a resource arbitration framework that adaptively prioritizes and schedules progressive iterative analytic jobs in a resource-limited environment. The framework

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00166



(a) Online aggregation progress of Query 5, 7, 19 of TPC-H. The percentage of data processed achieves 100% when the queries received and processed the entire TPC-H dataset (SF=1) in batches from a data source.



(b) Evaluation accuracy of five well-tuned popular convolutional neural network models on CIFAR-10 with batch 128 and learning rate 0.01.

Fig. 1: Progress curves of AQP and DLT jobs

can interrupt (or preempt) a currently running job in favor of another based on progress introspection and estimation. For instance, for some short-running jobs expected to achieve substantial progress and complete quickly, Rotary can preempt resources to process them instead of some long-running jobs. *The need for a resource arbitration framework arises for two reasons: (1) from the perspective of single jobs, it is reasonable to sacrifice precision for a quicker result; (2) from the perspective of the overall workload, it is beneficial to dynamically allocate and preempt resources to different jobs, for example, giving more resources to more promising jobs and constraining the resources for jobs stop progressing.*

We believe that resource arbitration and traditional scheduling systems [10]–[22] solve different but complementary problems. Scheduling systems are generally designed to optimize the execution and placement of the jobs according to the users' resource requirements and ensure the jobs can be completed on time. By contrast, resource arbitration systems are responsible for continuous resource allocation and preemption, determining when to start (or resume) and stop (or checkpoint) the progressive iterative analytic jobs based on the processing progress, available real-time resources, and users' completion criteria. In particular, as shown in Fig. 2, resource arbitration must consider a job's completion criteria and respond adaptively – something no prior scheduling system does. For example, consider an application scenario of hyperparameter optimization [23] for deep learning models, where a set of hyperparameter configurations are sampled from a hyperparameter space and formed a number of training trails that run iteratively and keep returning intermediate training results. Such a process is executed repeatedly until the best-performed hyperparameter configurations are selected. Thus, resource arbitration could stop the trials that contain unpromising hyperparameter configurations prematurely and allocate more resources to the promising ones so that the best-performing hyperparameters can be discovered sooner.

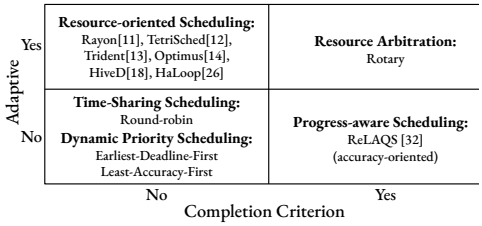


Fig. 2: Work Positioning

To realize this framework, we implement two prototype systems, *Rotary-AQP* and *Rotary-DLT*, for approximate query processing and deep learning training applications. For Rotary-AQP, we first extend a single-user progressive query processing system based on Apache Spark [24] and modify it to a multi-tenant AQP system. Then, we build the resource arbitration system on top of the multi-tenant AQP system. We evaluate Rotary-AQP using the TPC-H benchmark, and the evaluation results show that Rotary-AQP outperforms the state-of-the-art system and other baselines by allowing more TPC-H queries to reach their goals within the same amount of time. For Rotary-DLT, we build the system on top of TensorFlow [25] and conduct an evaluation using the workloads derived from a survey of 30 deep learning researchers across multiple research organizations. The evaluation results demonstrate that Rotary-DLT is superior to three dynamic priority-based baselines across a variety of optimization objectives. The two system implementations and their outstanding performance confirm the generality and practicality of our resource arbitration framework.

To summarize, our primary contributions include: (1) defining resource arbitration as a novel and specialized scheduling paradigm for progressive iterative analytic applications; (2) proposing a general resource arbitration framework, Rotary, and a new cost model that leverages the estimation of progress and resource consumption for job prioritization and preemption; (3) implementing two resource arbitration systems for approximate query processing and deep learning training, following the proposed framework.

II. Related Work

To the best of our knowledge, Rotary is the first resource arbitration system for AQP and DLT. Thus, we broadly review the related works and position our work.

A. Scheduling for Data Analytics

The existing iterative data processing scheduling works are related to our framework. One of the most important early works in this area is HaLoop [26]. The goal of HaLoop is to support iterative programs on Hadoop and optimize their execution by making the task scheduler loop-aware and by adding various caching mechanisms. Extending the line of HaLoop, SQLoop [27] is proposed to allow SQL to express iterative computations and schedule the SQL execution for potential parallel execution.

Progressive data processing, which is best exemplified by approximate query processing [28], [29], is also relevant to our resource arbitration framework. However, there is a lack of work exploring adaptive job preemption based on diverse completion criteria. One approach, iOLAP [30], returns intermediate results by processing the input data a batch at a time rather than running the query on the entire dataset. iOLAP partitions the input data into mini-batches, schedules the delta update query on each batch, and collects query results. It also can schedule recomputing jobs to recover the query result when a failure is detected. S-AQP [31] is similar work to iOLAP lies in this area. However, they mainly focus on scheduling query plans.

For scheduling AQP jobs, ReLAQS [32], which serves as one of the baselines in our experiments, is state-of-the-art. It can preempt the AQP jobs according to the estimation and try to help more jobs achieve their objectives. However, our framework has additional contributions: (1) Rotary is a general framework that can apply to other areas, such as deep learning training; (2) ReLAQS only schedules CPU cores, Rotary-AQP, our implementation for AQP under Rotary, further considers memory consumption when preempting resources; (3) Compared with ReLAQS, Rotary-AQP can support adaptive running cycling for short-running and long-running AQP jobs; (4) Estimation of ReLAQS only uses real-time results to predict the progress of each AQP job for the next running epoch, the estimators in Rotary-AQP jointly utilize historical and real-time data to make predication which can overcome some issues such as cold-start or data bias.

B. Scheduling for Machine Learning

We consider scheduling systems for machine learning as related works. MArk [33] allows users to specify the response time for machine learning model serving and schedules by selecting between AWS EC2 and AWS Lambda to support unpredictable workload bursts. Some works like Tiresias [16] and Optimus [14] schedule machine learning jobs with time constraints. Gandiva is a cluster scheduling framework that utilizes the cyclic predictability of intra-batch in a DLT job and the feedback of early training to improve training latency and efficiency in a GPU cluster [15]. Philly analyzes a trace of machine learning workloads run on a cluster of GPUs in Microsoft and schedules the jobs according to a trade-off between locality and GPU utilization [17]. HiveD [18] is designed to be a Kubernetes scheduler extension for Multi-tenant GPU clusters, which can guarantee resource reservation for DLT jobs. PipeDream [34] is a deep learning training system that schedules computation by pipelining execution across multiple machines to accelerate the training process. AntMan [35] is a large-scale deep learning multi-tenant infrastructure in Alibaba, which utilizes the spare GPU resources to co-execute multiple jobs on a shared GPU and dynamically scales memory and computation. Pollux [20] is resource-adaptive deep learning (DL) training and scheduling framework which optimizes inter-dependent factors both at the per-job level and at the cluster-wide level.

As we emphasized before, scheduling systems and our resource arbitration framework, Rotary, solve different but complementary problems. The scheduling systems pay more attention to resource reservation and job placement according to jobs' requirements. However, Rotary addressed the issues about adaptive resource allocation and job preemption for diverse completion criteria.

C. Priority Scheduling

Priority scheduling is a method of scheduling processes that is based on priority, which is widely used in operating systems for CPU scheduling. These scheduling methods are either non-preemptive or preemptive [36]. Rotary shares a similar principle with preemptive priority scheduling: we should interrupt a lower-priority task in favor of some task with a higher priority if it is beneficial to do so. There have been plenty of works in this area over the past decades [37]–[40].

Although Rotary sounds analogous to preemptive priority scheduling if the completion criteria are treated as custom-defined priorities, the crucial differences between them are: 1) task priorities in preemptive priority scheduling generally is fixed, while Rotary considers priorities of each task is dynamic according to various completion criteria and diminishing returns; 2) Rotary can model the change of the task priorities in the resource arbitration framework.

III. Resource Arbitration Framework

A. Terminology and Setup

First, we define a common set of terms to describe progressive iterative analytic jobs. In a progressive iterative analytic job, data are processed in batches, where each *batch* is a subset of the entire dataset or a data stream that is progressively sampled from the overall data, each batch has the (approximately) same batch size. A progressive iterative analytic job moves one *step* when it finishes processing a single batch. After a fixed number of such steps (called an *epoch*), the job's performance can be evaluated based on *convergence metrics* on the returned results. Convergence metrics are usually some proxy for result accuracy. One progressive iterative analytic job typically runs for multiple epochs until the user is satisfied with its convergence or reaches some user-defined completion criteria such as running time.

Example 1. Approximate Query Processing in SQL: Approximate query processing can provide quick, approximate results to users by running queries on a subset of the overall dataset or a data stream. One technique to realize AQP is online aggregation [28]. Online aggregation systems process data iteratively using data batches, and each progressive sampling of the data is a batch and processes roughly the same amount of data, as they are each of approximately the same size. Online aggregation systems calculate error bounds, such as confidence intervals, after each batch is processed so that users can decide whether to continue processing. In AQP, a batch and an epoch can be synonymous, and the convergence metric is the size of the confidence interval.

Example 2. Deep Learning Training: A typical DLT job consists of a neural network model (e.g., ResNet [41] or Bi-LSTM [42]), a dataset for training and evaluation, and a set of hyperparameters (e.g., batch size, learning rate, optimizer, etc.). During the training process, the training dataset is iteratively sampled in batches, and each training step is one optimization step updating the parameters (or gradients) of the neural network model based on the batch. In the context of DLT, an epoch normally is a complete pass of the training data. Once the neural network model has been trained for one epoch, it will be evaluated on the evaluation dataset in terms of convergence metrics, which can be either training loss or validation set accuracy. This process is applied repeatedly until the desired convergence target is achieved. As models have become more complex, DLT largely relies on specialized hardware devices like GPUs and TPUs.

B. User-defined Completion Criteria

Rotary allows users to define their own completion criteria, and herein we take three types of completion criteria based on the common practice of DLT and AQP as examples. As presented in Fig. 3, there are ❶ *accuracy*-oriented completion criteria, ❷ *convergence*-oriented completion criteria, and ❸ *runtime*-oriented completion criteria. Essentially, such completion criteria are add-ons to the regular query and training commands and should be orthogonal to the execution of AQP and DLT without modifying the original command parsers.

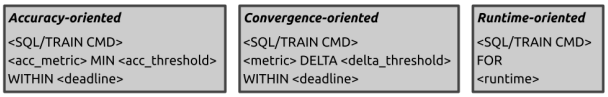


Fig. 3: Templates of user-defined completion criteria

Fig. 4 shows three completion criteria examples following the templates. The left one illustrates how to add a completion criterion of achieving at least 95% accuracy within 3600 seconds, the middle one defines a completion criterion for training a ResNet model until reaching the convergence of 0.001 within 30 epochs, the right one will train the MobileNet model for 2 hours and return the training results anyway.

Fig. 4: Examples of user-defined completion criteria

❶ Accuracy-oriented completion criteria are widely used and allow users to explicitly specify an expected accuracy within maximum training epochs. In the above example, we use *ACC* (i.e., training accuracy), which is a common metric, but other user-defined metrics, such as F1 score and Perplexity, are supported as well. Additional error bounds, such as confidence interval, are optional as well. The deadline could be expressed in epochs or time units.

❷ Convergence-oriented completion criteria are also typical, especially for DLT jobs. With these criteria, a job is considered "complete" once its performance is found to no longer increase. In the middle example of Fig. 4, *ACC* is used for measuring convergence, but other metrics, such as *LOSS* [43], can be used for convergence. The convergence-oriented criteria also allow users to specify a deadline, which means a job will be terminated if it fails to converge until the deadline.

❸ Runtime-oriented completion criteria are proposed for users who want to execute their progressive iterative analytic jobs for a while without any explicit objective or threshold. As the *WITHIN* predicate we have in the other two completion criteria, the runtime can be the number of epochs or a period of time, such as training a model for 100 epochs or running a query for 6 hours.

C. Framework Architecture

We identify three opportunities to address the resource arbitration problem.

First, the diverse completion criteria of progressive iterative analytics bring the opportunity to allocate various amounts of resources to different jobs while still achieving their objectives. For example, it makes more sense to give fewer resources to a job that only needs to achieve an effortless objective.

Second, diminishing returns of progressive iterative analytic jobs indicates that the value of two data batches to a user may be completely different. This makes iterative resource allocation and preemption practical and valuable. For instance, it may be beneficial in some situations when a data batch that provides more valuable results to users can be processed completely sooner if more resources are allocated continuously. This leads to a cost model which should balance the progress improvement (i.e., providing more valuable results) and resource consumption (the cost to improve the progress or produce the results). An example of this can be seen in Fig. 1b, where we show that the earlier training epochs could improve the deep learning models' accuracy more significantly than the older ones, and the users could get a decent trained model more quickly if more resources are given to the jobs with more potential for improvement; however, the trade-off between performance improvement and the models' GPU memory requirements need to be addressed as well.

Third, different data processing rate of progressive iterative analytic jobs rationalizes the adaptive running epochs; namely, long-running jobs should be allowed to have a longer running epoch after arbitrating and allocating resources so that they can return expected intermediate results. This can be exemplified by Fig. 1a, where we present that the process of query 19 increases more expeditiously than queries 7 and 19 when they are all checked every 60 seconds; however, we can observe all the queries will have a similar pattern of progress improvement if query 5 and query 7 check every 120 and 180 seconds.

To exploit these opportunities, we proposed the resource arbitration framework, Rotary. We show the framework's architecture and highlight the core components in Fig. 5. Rotary allows users to submit their progressive iterative analytic jobs along with the corresponding completion criteria. Once submitted, Rotary considers these jobs active and is ready to run them. Rotary's engine is responsible for resource arbitration. It can estimate how much progress a job can achieve in terms of completion criteria and how many resources the job will

consume for such progress. Rotary can prioritize jobs according to a cost model and arbitrate the resources for them. Once the process of resource arbitration is finished, the selected jobs will be deployed in Rotary execution, where the resource is allocated and preempted to the jobs so that they can run in an execution platform (e.g., PyTorch or TensorFlow for deep learning, Spark for query processing). When the jobs complete the current epoch, they can be checkpointed or materialized if they are not granted resources for the next running epoch. Furthermore, it is beneficial to store the progressive iterative analytic jobs and track intermediate processing results since such information can be used to provide a better estimation.

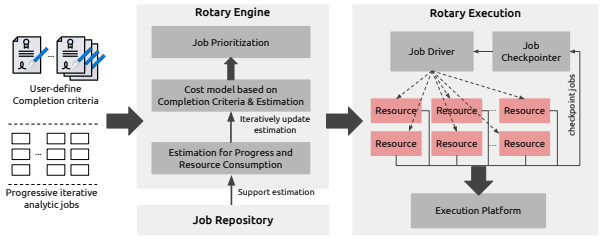


Fig. 5: Framework architecture of Rotary

Rotary also can re-evaluate and schedule the jobs that have been deferred or are currently running for the next epoch. The advantages of this ability are three-fold. First, it provides the resource arbitration system with a wider range of running options for progressive iterative analytic jobs compared with the systems that exclusively consider the current jobs. Second, the deferred jobs can be reconsidered for running when it is beneficial to do so, which can prevent them from waiting for an unexpectedly long time. Third, the overhead of job interruption, such as checkpointing to disk, can be avoided if a job is continuously prioritized by Rotary.

D. Resource Arbitration Problem Statement

Workloads. Consider a workload W that consists of n progressive iterative analytic jobs $\{j_1, \cdots, j_n\}$, each job processes data batch-by-batch and returns the intermediate processing results for every epoch. Each i^{th} job emits a time-series per-epoch intermediate state $\{ins_{(i,0)}, ins_{(i,1)}, ..., ins_{(i,t),...}\}$ which contains the convergence results and attainment progress ϕ toward its specific user-defined completion criterion c. Thus, there is a list of criteria $C = \{c_1, \cdots, c_n\}$ associated with jobs in the workload W. Each job in the workload will terminate if $c(ins_{(i,t)}) == \texttt{true}$. Once a job w reaches its completion criteria, it is de-queued $W = W \setminus w$.

Resources. These jobs have to be assigned to a particular "computing resource" (e.g., an available GPU or CPU hardware thread). There are M such resources considered, and they are possibly heterogeneous. These resources can only process one job at a time and are not sub-dividable. A job holds on to a particular resource for at least an epoch. Thus, at any given time, the current resource usage can be modeled as a bi-partite assignment where a subset of jobs are mapped to unique resources $\texttt{assign}(W, M)$. As these assignments change, jobs have to be loaded to the resources and checkpointed accordingly.

Resource Arbitration Policy. A resource arbitration policy is a function that produces assignment decisions (and interrupts previous assignments if necessary) based on the current state of the queue Q_t, which is the intermediate state associated with completion criteria of all the jobs currently in the queue.

$$\pi : Q_t \mapsto \texttt{assign}(W, M)$$

The application of this policy results in a sequence of resource assignment decisions at each time-step.

Objective. At each epoch t, attainment progress $\phi^t_{j_i}$ denotes the progress of job j_i toward its completion criteria, $A_t = n - |W|$ quantifies the number of jobs that reach their completion criteria (i.e., $\phi^t_{j_i} = 100\%$), which is further exploited to denote the workload's attainment rate $\psi_t = \frac{A_t}{n}$. The objective of Rotary is to maximize a utility function that can be constrained by fairness and efficiency. If fairness is the objective, Rotary will maximize $\min \phi_{j_i}, 1 \leq i \leq n$ and keep allocating resources to the job with the lowest job attainment progress. If efficiency is prioritized, Rotary will maximize ψ by continuously selecting the jobs that can achieve higher attainment progress.

E. Resource Arbitration Algorithm

We propose an algorithm sketch for addressing the problem, as presented in Algorithm 1. For each epoch, the jobs that are selected to run may achieve different attainment progress toward their completion criteria. Suppose the completeness of each job can be treated as a job priority. In that case, such dynamic "job priority" requires Rotary to timely capture the current attainment progress of each job (especially for the ones with diminishing returns) and adaptively estimate the "priority" for them in each epoch based on the current progress, estimated future progress, and their diverse completion criteria, so that the most appropriate jobs can be selected for next running epoch.

Algorithm 1: Algorithm Sketch for Resource Arbitration

```
while not all jobs reach completion criteria do
    for jobs is active but not attained j_i, i ← 1 to n do
        Estimate the attainment progress φ̂_{j_i} for next epoch;
    Resource arbitration for active jobs based on {φ̂_{j_i} | ∀i = 1..n};
    for selected jobs do
        Executing the selected job;
        Observe the attainment progress for the selected job;
```

However, the system implementations for various applications may have different algorithms to address the problem. Following the algorithm sketch, we design two algorithms for AQP (§IV-A) and DLT (§IV-B).

IV. System Implementation

Following the proposed framework, we illustrate how we implement the resource arbitration prototype system for approximate query processing (Rotary-AQP) and deep learning training (Rotary-DLT) and further discuss their similarities and differences.

A. Rotary-AQP Implementation

To implement Rotary-AQP, we modify a single-user progressive query processing system based on Apache Spark [24] and make it a multi-tenant environment by adding concurrency control and checkpoint mechanisms. This system serves as our execution platform to run the AQP jobs.

Rotary-AQP can take AQP jobs with pre-defined completion criteria. We take accuracy-oriented completion criteria (a widely-used metric in AQP [4], [30], [32], [44]) as examples. Specifically, each job is attached with an accuracy threshold and a deadline to reach the threshold, thus the processing progress in the framework is measured in terms of accuracy in this implementation of AQP. Rotary-AQP processes AQP jobs and arbitrates the resources for them so that more jobs can reach their accuracy threshold. Rotary-AQP focuses on online aggregation [45]. The accuracy of aggregation is calculated as $accuracy = \frac{\alpha_c}{\alpha_f}$, where α_c is the current aggregation result, and α_f is the final aggregation result. Considering the aggregation operations are column-oriented, the accuracy of an AQP job that performs multiple aggregation operations on multiple columns can be calculated as $accuracy = \frac{1}{k}\sum_{i=0}^{k} \alpha_c^k/\alpha_f^k$, where α_c^k is the current aggregation result on column k and α_f^k is the final aggregation result on column k. This is based on the assumption that all columns are of equal importance (which is applied to our evaluation). However, Rotary-AQP also allows the users to specify the importance of each column by assigning weights.

We use a non-parametric confidence interval estimator to assess convergence. The technique is based on envelope functions from empirical process theory [23]. Rotary-AQP keeps tracking the least and largest aggregation results within a time window (e.g., t epochs) and uses this gap to determine convergence[1]. Given that the aggregation will eventually converge, the gap between the least aggregation result (denoted by p) and the largest aggregation result (denoted by q) can be substantial but should be shrunk gradually over time. Thus, the accuracy progress can be expressed as $\frac{p}{q}$, which can provide an approximate estimate for the accuracy progress of an aggregation operation in the AQP jobs.

Following the architecture in Figure 5, Rotary-AQP has two core components for estimating the accuracy progress and memory consumption. The accuracy progress estimator is used for prioritizing jobs. It estimates the potential accuracy of a job j for the next epoch if the resources are granted. Its core idea is to fit a progress-runtime curve leveraging historical and real-time data. The historical data are from the selected historical jobs that are similar to job j according to query features such as query predicates, query table and column names, and query batch size [46]. The real-time data can be conveniently obtained since Rotary-AQP tracks the running AQP jobs. We further exploit weighted linear regression [47] to learn a curve for estimation based on the collected historical and real-time data. Specifically, the estimator selects top-k similar historical jobs for an AQP job to fit an initial progress-runtime curve that can be used for the first estimation. Then, when the job is placed and launched, Rotary-AQP records the real-time intermediate aggregation results and continuously adjusts the fitted curve by adding these real-time results. Due to the importance of real-time results, each recorded real-time result and the combination of all the historical data will share equal weight when fitting the progress-runtime curve. For instance, if one recorded real-time result and a number of selected historical data are used to fit a curve, the real-time result will be granted 0.5 weight, and all the historical data as a whole will get the remaining 0.5 weight. By extension, when three real-time intermediate results are recorded for fitting the curve, each result and the combination of all the historical data will share 0.25 weight, respectively. This continuous joint fitting method makes the estimated progress-runtime curve reasonably close to the ground truth and sufficient for estimating the progress.

The memory consumption estimator can make sure there will be sufficient memory to support jobs. It predicts the memory consumption of the AQP jobs based on each batch's table and column statistics and query plans in the AQP, which has been well-studied. In our implementation, we exploit Apache Spark's CBO [48] to obtain memory consumption before running the AQP jobs. Rotary-AQP also tracks the number of table rows scanned, filtered, and aggregated. The memory consumption estimator also supports adaptive running epochs by determining the length of the running epoch (e.g., the number of batches in an epoch). We observe that the AQP jobs that consume larger memory usually take a (proportionally) longer time to process a batch, and these jobs deserve a longer running epoch accordingly. Thus, Rotary-AQP makes the length of the running epoch of every AQP job proportionate to the estimated memory consumption.

Rotary-AQP can arbitrate computing resources for the jobs based on estimated accuracy progress and memory consumption, as presented in Algorithm 2. During each epoch, Rotary-AQP will first allocate one hardware thread to each active job that can fit in memory. Rotary-AQP further ranks these active jobs and allocates extra computing resources to the ones that can achieve higher progress toward their completion criteria.

B. Rotary-DLT Implementation

Rotary-DLT follows the architecture in Figure 5. Compared to Rotary-AQP, Rotary-DLT has the following differences: (1) a training epoch estimator (*TEE*) to predict the number of training epochs to achieve a specific accuracy, and a training memory estimator (*TME*) to predict the memory usage of a deep learning model; (2) a training time recorder (*TTR*) to measure the time of a training epoch; (3) GPU resource arbitration for the DLT jobs; and (4) TensorFlow is deployed as the execution platform to run the DLT jobs. Furthermore, Rotary-DLT stores the information of the historical DLT jobs in a repository so that the system can provide more accurate estimates for attainment progress and memory consumption. All the completed jobs' information are stored, including

[1]The formal derivation of this estimator has been cut for brevity.

Algorithm 2: Resource Arbitration for AQP

Input : Workload $W = \{j_1, \cdots, j_n\}$
Completion Criteria $C = \{c_1, \cdots, c_n\}$
Total CPU hardware threads D, Total memory M

//All jobs are placed in an active queue when arriving
for *job* $j_i \in W$ *that arrives* **do**
  Mark job j_i as active and place it to the active queue AQ;

//Resource arbitration for the jobs in the waiting queue
while $AQ \neq \varnothing$ **do**
  Initialize priority queue PQ;
  for *active jobs* j_i, $i \leftarrow 1$ **to** n **do**
    Estimate the memory consumption $\hat{m}_{j_i}$;
    Assign running epoch e_{j_i} for job j_i;
    Estimate the progress $\hat{\phi}_{j_i}$ toward their completion criteria;
    Place job j_i in PQ due to $\hat{\phi}_{j_i}$;
  ResourceArbitration (*active jobs*) ;
  Run active jobs, and mark them as running;
  for *active jobs* j_i, $i \leftarrow 1$ **to** n **do**
    if j_i *finish one epoch* e_{j_i} **then**
      Observe the accuracy progress ϕ_{j_i} for current epoch;
      if *job* j_i *meets* c_{j_i} **then** remove from AQ ;
      Mark job j_i as active;

Function ResourceArbitration (*jobs*) **:**
  for *job* j_k *in jobs* **do**
    if $\hat{m}_{j_k} \leq M$ **then**
      Allocate 1 hardware thread to job j_k;
      $D = D - 1$, $M = M - \hat{m}_{j_k}$;
    else
      if *job* j_k *in* PQ **then** remove j_k from PQ ;
  for *job* j_k *in* PQ *&* $D \neq 0$ **do**
    Allocate extra 1 hardware thread to job j_k, $D = D - 1$;

model architecture, training hyperparameters, training epochs, and evaluation accuracy.

A key feature of Rotary-DLT is to estimate the number of epochs for training DLT jobs to achieve specific accuracy, which is accomplished by TEE. Considering that DLT jobs always center on the accuracy metric, TEE is beneficial for Rotary-DLT to know whether it should allocate or preempt resources for the scheduled jobs. When estimating the number of needed epochs for job j to achieve a specific training accuracy, TEE first selects top-k similar historical DLT jobs to job j in terms of the metadata of training dataset and training hyperparameters such as learning rate, training batch size, and optimizer [49], and then extract the data pair (accuracy, epoch) from the historical job. TEE also captures the pair (accuracy, epoch) during the training process of job j. Similar to the progress estimator of Rotary-AQP, TEE fits an accuracy-epoch curve by jointly using historical and real-time data using weighted linear regression, and every recorded real-time data and the combination of the historical data share equal weight.

TME is another key component and can predict the maximum GPU memory usage of the models in the jobs so that DLT jobs can be launched on a target GPU with sufficient memory. As we mentioned in §III-A, the training batch size remains the same for each training iteration, and it directly decides how much data will be transferred from the host memory to GPU during each batch. Moreover, all the learnable parameters in a deep learning model require space in memory, and these parameters where historic gradients are being calculated and used also accumulate in memory. Thus, it is viable to estimate the memory usage of deep learning models if the training batch size and model parameter information are given. We fit a batch size-memory curve by leveraging the data from historical jobs for TME. When estimating the memory usage of a DLT job, TME first retrieves all the data of historical jobs that use the same training dataset and then computes the similarity between the target job and the historical jobs. The similarity between the two jobs is defined as $similarity(x, y) = 1 - \frac{|x-y|}{max(x,y)}$, where x and y are the numbers of model parameters (i.e., model size) of the two jobs, respectively. Afterward, TME picks top-k similar historical jobs to fit the batch size-memory curve. We also exploit the weighted linear regression to fit the curve but in a different way: the more similar a historical job is, the higher weights the job will be granted. Furthermore, we pad the estimated memory by an additional offset to minimize the likelihood of out-of-memory (OOM) issues.

There are two fundamental differences between AQP and DLT, which should be considered for implementing Rotary-DLT. First, DLT jobs can be evaluated every one or multiple epochs using an evaluation dataset; thus, it is unnecessary for Rotary-DLT to have a mechanism like an envelope function in Rotary-AQP to approximately evaluate the progress of each job. Second, the batch processing time of AQP jobs can be quite different due to the query predicates and heterogeneous data batch; for example, a batch may trigger numerous join and aggregation operations, but the others may not. However, DLT jobs usually have similar batch processing time due to the stable model architecture and the same batch size. Thus, Rotary-DLT has a side component, TTR, to record the training time of a single step or an epoch. TTR records the time of a training step or a training epoch for each DLT job on different devices to reduce the recording overhead. Due to a CUDA warm-up issue [49], the very first training step always takes a longer time, so we always discard the first training step when TTR is running and recording. Since the deep learning training job is launched in iteration, recording the single training time of each job is sufficient to measure the overall time of the training process.

Following the problem statement (§III-D), we devise a threshold-based resource arbitration algorithm for DLT (Algorithm 3). As an example to balance fairness and efficiency, this algorithm can prioritize various jobs by allocating/preempting the available GPU resources according to a threshold T. More specifically, for each epoch, the algorithm will prioritize the jobs with the lowest attainment progress until all the jobs either achieve T progress or are considered converged so that no single job will considerably fall behind; then, the algorithm will continuously select the more promising jobs that can achieve higher progress in a relatively shorter period so that more jobs can be completed quickly. Therefore, when $T = 0\%$, the algorithm is always efficiency-oriented since every job achieves at least 0% progress from the beginning, so the algorithm aims to achieve a higher attainment rate for a workload. If $T = 100\%$, the algorithm is fairness-oriented

Algorithm 3: Adaptive Resource Arbitration for DLT

Input : Workload $W = \{j_1, \cdots, j_n\}$
Completion Criteria $C = \{c_1, \cdots, c_n\}$
Total GPU D, GPU memory $\{M_1, \cdots, M_D\}$

//All jobs are placed in an active queue when arriving
for *job* $j_i \in W$ *that arrives* **do**
  Place job j_i to the active queue AQ
while $AQ \neq \varnothing$ **do**
  if *all jobs from* W *meet* T **then**
    Create a queue PQ that prioritizes highest progress job;
  else
    Create a queue PQ that prioritizes lowest progress job;
  for $i \leftarrow 1$ **to** n **do**
    Estimate the resource consumption $\hat{m}_{j_i}$ for job j_i;
    Estimate the training progress $\hat{\phi}_{j_i}$ for job j_i;
    Place job j_i in AQ according to $\hat{\phi}_{j_i}$
  for $d \leftarrow 1$ **to** D **do**
    for *job* j_k *in* AQ **do**
      if $m_{j_k} \leq M_d$ **then**
        Run job j_k on GPU d, Remove job j_k from PQ;
  for *job in* AQ *achieves the completion criteria* **do**
    Remove job from AQ;

since it keeps allocating resources to the jobs with the lowest attainment progress until all the jobs are finished. By tuning the threshold, the proposed resource arbitration algorithm can tweak fairness and efficiency.

As a core in the resource arbitration algorithm for DLT, the computation of training progress ϕ differs for various completion criteria. For example, for the jobs with runtime-oriented completion criteria, calculating ϕ is trivial, which is the ratio of current runtime (e.g., number of epochs) to the runtime threshold. For the jobs with accuracy-oriented and convergence-oriented completion criteria, ϕ can be obtained by estimating the current accuracy and comparing it with the target accuracy. We present the computation of training progress in Algorithm 4.

Algorithm 4: Progress Computation in Rotary-DLT

Input : Workload $W = \{j_1, \cdots, j_n\}$
Completion Criteria $C = \{c_1, \cdots, c_n\}$
for $i \leftarrow 1$ **to** n **do**
  $e_i^* \leftarrow$ job running progress (training epochs) of j_i;
  if j_i *has runtime-oriented completion criteria* **then**
    Obtain expected training epoch e_i according to c_i;
    $\phi_i = \frac{e_i^*}{e_i}$;
  else if j_i *has accuracy-oriented completion criteria* **then**
    Obtain maximum training epoch e_i^{max} according to c_i;
    Estimate the necessary epochs $\hat{e}_i$ according to c_i;
    if $\hat{e}_i > e_i^*$ **then** $\phi_i = \frac{e_i^*}{e_i^{max}}$ **else** $\phi_i = \frac{e^*}{\hat{e}_i}$;
  else if j_i *has convergence-oriented completion criteria* **then**
    Obtain maximum training epoch e_i^{max} according to c_i;
    Obtain expected accuracy acc_i according to s_i;
    Estimate the necessary epochs $\hat{e}_i$ according to acc_i;
    if $\hat{e}_i > e_i^*$ **then** $\phi_i = \frac{e_i^*}{e_i^{max}}$ **else** $\phi_i = \frac{e^*}{c_i}$;

V. Evaluation

We evaluate our two Rotary prototype systems, Rotary-AQP and Rotary-DLT, respectively.

A. Rotary-AQP Evaluation

Our evaluation for Rotary-AQP addresses the following questions:

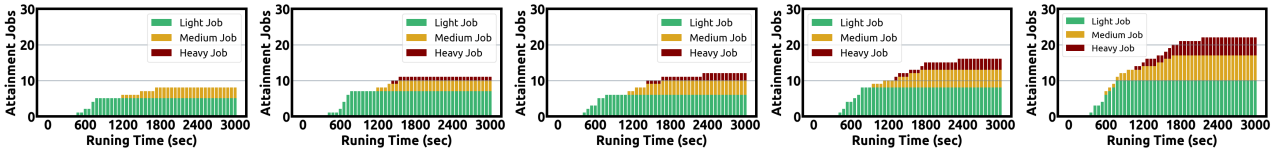
- Can resource arbitration improve the number of jobs that attain their performance objective, compared to a state-of-the-art approach and other common baselines? (§V-A2)
- What is the overhead of resource arbitration? (§V-A3)
- How does the distribution of job resource requirements impact the performance of Rotary-AQP? (§V-A4)
- How does the progress estimation impact the performance of Rotary-AQP? (§V-A5)

All the experiments are conducted on a server with two Intel Xeon Silver CPUs (2.10GHz, 12 physical cores) and 192GB memory, running Ubuntu Server 18.04. For all experiments, we use 20 physical cores and leave the rest for the OS (Ubuntu 18.04). We use an Apache Kafka [50] cluster on a different machine with the same hardware configuration as the data source for AQP queries.

We implement four baselines for comparison: ReLAQS [32], EDF (Earliest Deadline First), LAF (Least Accuracy First), and round-robin. As a vanilla baseline, round-robin allocates one core to each job in turn until there are no more cores and run them for an epoch per time until they reach their completion criteria (either achieve the accuracy threshold or beyond the deadline). EDF and LAF are two dynamic priority-based baselines that always prioritize the jobs that have the earliest deadline and least accuracy, respectively. ReLAQS is the state-of-the-art work, which is a multi-tenant system for AQP that aims to reduce the average latency of a workload by scheduling CPU cores to jobs with the most potential for improvement. In ReLAQS, the potential improvement of each job is simply estimated according to previous processing results. Compared with ReLAQS, Rotary-AQP considers both CPU and memory for resource arbitration, combines historical and real-time data to estimate the accuracy progress, and supports adaptive running epochs for short-running and long-running AQP jobs.

*1) **AQP Workload**:* We evaluate Rotary-AQP using the TPC-H benchmark. Rotary-AQP supports all 22 queries and runs them on the TPC-H dataset. Given the number of concurrent jobs and Spark's in-memory requirement, we limit the scale factor to 1. Larger scale factors should not affect the performance of Rotary-AQP but require more memory to run multiple AQP jobs simultaneously.

The workload consists of 30 AQP jobs, each of which is a random query selected from the 22 TPC-H queries. According to the observed memory consumption of queries, we categorize the TPC-H queries into three groups: light, medium, and heavy queries. The workload is a mixed collection of jobs for the queries from the three groups, and the proportions of the jobs in the three groups can be adjusted. In the workload, each job is attached with an accuracy threshold and deadline, which are both randomly selected from two parameter spaces. Furthermore, to simulate users submitting approximate queries to the shared cluster, the job arrives according to a Poisson



(a) Round-robin (b) EDF (c) LAF (d) ReLAQS (e) Rotary-AQP

Fig. 6: Evaluation of Rotary-AQP and four baselines (Round-robin, EDF, LAF, ReLAQS) on the synthetic AQP workload

Queries	Light	q1, q2, q4, q6, q10, q11, q12, q13, q14, q15, q16, q19, q22
	Medium	q3, q5, q8, q17, q20
	Heavy	q7, q9, q18, q21
Completion Criteria	Accuracy	55%, 60%, 65%, 70%, 75%, 80%, 85%, 90%, 95%
	Deadline	Light Queries Deadline (sec): 360, 420, 480, 540, 600, 660, 720, 780, 840, 900 Medium Queries Deadline (sec): 1080, 1200, 1320, 1440, 1560, 1680, 1800, 1920, 2040, 2160 Heavy Queries Deadline (sec): 1440, 1620, 1800, 1980, 2160, 2340, 2520, 2700, 2880, 3060
Workload	40% AQP jobs with light queries 30% AQP jobs with medium queries 30% AQP jobs with heavy queries	

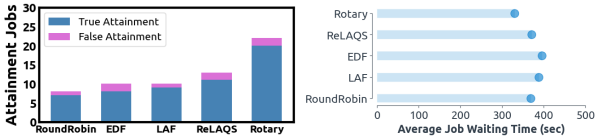
TABLE I: Synthetic AQP workload. The selection of query type, accuracy threshold, and deadline, are all random and based on a uniform distribution. Job arrival is based on a Poisson distribution.

distribution with a mean arrival time of 160 seconds. The configurations of the workload are elaborated in Table I.

*2) **Attainment for AQP Workload**:* Attainment rate serves as the most important benchmark since it measures how many jobs reach their accuracy threshold, namely, users are satisfied with the results. Fig. 6 shows the overall number of attained jobs (e.g., jobs that met their convergence criteria before their deadline) under Rotary-AQP, which exceeds those using the four baselines. Although Rotary-AQP can attain more jobs for light, medium, and heavy queries, it performs best for jobs with heavy queries. This is mainly due to two reasons. First, Rotary-AQP can provide better progress estimation by jointly leveraging historical and real-time data to find the jobs with the most potential for improvement. Second, Rotary-AQP can give the proportional running epochs to various jobs according to their job size (i.e., estimate memory consumption in the implementation). Thus, heavy jobs, often long-running, can return progressive results and be fairly compared with short-running jobs during resource arbitration. Therefore, compared with the baselines, Rotary-AQP allows the heavy jobs to have a higher chance to gain more resources for running. Such results confirm the efficiency and effectiveness of Rotary-AQP.

*3) **False Attainment and Waiting Time**:* We use an envelope function to determine when to stop the jobs, but the envelope function can make mistakes, such as stopping the jobs which are not supposed to be permanently terminated, which we consider as false attainment. We present the false attainment for Rotary-AQP and the baselines in Fig. 7a. The envelope function can provide reliable decisions generally but still make mistakes. This issue can be mitigated by lengthening the time window of the envelope function.

We also tally the average waiting time of the jobs in the



(a) False Attainment (b) Average Waiting Time

Fig. 7: False attainment and waiting time of Rotary-AQP

workload, as shown in Fig. 7b. The waiting time of a single job is calculated as the difference between its running time under Rotary or other baselines and the time of running it independently and isolated. Our system also outperforms other baselines due to the adaptive running epochs. More specifically, unlike Rotary-AQP, other baselines are in favor of short-running jobs, which can achieve higher accuracy progress in a short time which may defer the heavy job far into the future and lead to an unexpectedly long waiting time for the long-running jobs.

*4) **Skewed Workload**:* To evaluate Rotary-AQP on a balanced workload, we have 40% jobs with light queries, 30% jobs with medium queries, and 30% jobs with heavy queries. However, it is also reasonable to fathom the performance of Rotary-AQP on the workload with various job distributions. For this, we deploy Rotary-AQP and the baselines in three "extreme" cases: the workloads only consist of jobs with light jobs, medium jobs, and heavy jobs.

As we can see from Fig. 8, Rotary-AQP can achieve the best performance for all three skewed workloads, especially in the workload that only contains heavy jobs. Rotary-AQP and ReLAQS can defeat other baselines due to progress estimation, whereas Rotary-AQP performs better than ReLAQS because Rotary-AQP can collect more accurate real-time intermediate results due to the adaptive running epochs to make more reliable progress estimation for the next epoch.

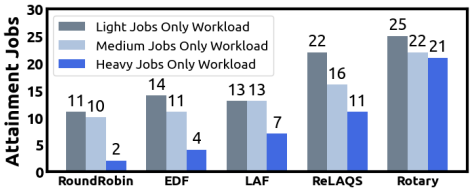
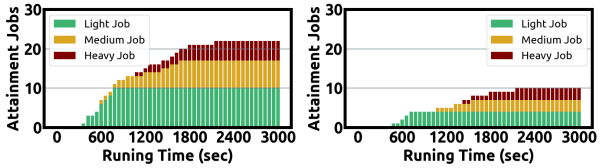


Fig. 8: Attained jobs in the various workloads (30 jobs)

*5) **Progress Estimation Sensitivity**:* Since the accuracy progress estimator serves as a core component of Rotary-AQP, we investigate how much it affects the performance of Rotary-AQP. Thus, we design a new baseline which is essentially Rotary-AQP, but their accuracy progress estimator will randomly return the estimated progress following a uniform distribution from 0 to 1. Such artificial progress estimation is misleading, and Rotary-AQP may make unwise resource

arbitration accordingly.

Fig. 9b displays the number of attained jobs under such artificial estimation, which is slightly better than round-robin (Fig. 6a) and almost tied to EDF (Fig. 6b) and LAF (Fig. 6c). The artificial estimation attains fewer light jobs than EDF and LAF but outperforms them according to the attainment rate of medium and heavy jobs. Such results indicate that (1) the accuracy progress estimator is vital to Rotary and (2) the adaptive running epochs can help some medium and heavy jobs to attain their goals.



(a) Rotary-AQP (b) Rotary-AQP with estimation

Fig. 9: Impact of progress estimation

B. Rotary-DLT Evaluation

We implement Rotary-DLT on top of TensorFlow 1.15 [25]. All the experiments are conducted on a server with Intel Xeon Silver CPU (2.10GHz), 192GB memory, and 4 GPUs (RTX 2080 8GB graphic memory), running Ubuntu Server 18.04. All the evaluation results are averaged over 3 independent runs.

*1) **Survey-based DLT Workload**:* To evaluate Rotary-DLT, we surveyed 30 experienced deep learning researchers across the following affiliations listed alphabetically: Microsoft Research, National University of Singapore, Northeastern University, Singapore Management University, University of California-Berkeley, University of Chicago, University of Illinois at Urbana-Champaign, and University of Toronto. According to their responses about training infrastructure, model architecture, running time, and completion criteria, we synthesize a DLT workload. The elaborate configurations of the synthetic workload are presented in Table II. We implement a number of representative deep learning models in Computer Vision (CV) and Natural Language Processing (NLP) with randomized hyperparameters and completion criteria. We use the small batch sizes to train the CV models due to the empirical study [51] but choose bigger sizes for NLP models due to common practice [52]. We follow the design in their original paper for other specific hyperparameters of some models (e.g., the growth rate for DenseNet). We also have pre-trained versions of BERT, VGG, and ResNet since the jobs of fine-tuning pre-trained models are also common.

For the models with multiple variants like ResNet, DenseNet, ShuffleNet, VGG, BERT, we use the shrunk variants (e.g., ResNet-18, ResNet-34, DenseNet-121) to fit them on a single GPU.

*2) **Attainment for DLT Workload**:* We consider fairness and efficiency as two vital but opposite optimization objectives. Achieving fairness can guarantee that no single job is stalled due to a myriad of jobs being in front of it or some upfront jobs taking an unexpectedly long time. Efficiency focuses on completing more jobs in a shorter time if possible, and this objective can only be achieved by always picking up the jobs that can be finished faster. If we stick with fairness, the jobs that can be completed quickly may have to wait a long time. On the contrary, concentrating on efficiency can result in zero progress in some jobs (they are never triggered).

We define three metrics of attainment progress for DLT jobs with various completion criteria.

- *Accuracy-oriented attainment progress*: Similar to attainment progress ϕ, this shows the completion percentage of a job with accuracy-oriented completion criteria but from the perspective of accuracy, which is defined as $\frac{current\ accuracy}{completion\ criteria}$. For instance, if a job has an accuracy target of 80% and obtains 56% accuracy after training one epoch. The current attainment progress is $\frac{56\%}{80\%} = 70\%$.
- *Convergence-oriented attainment progress*: We measure the attainment progress of jobs with convergence-oriented completion criteria in terms of epochs. When retrospecting the training process, if the jobs converged before the max training epochs, we mark the epoch as the *convergence-line* when the model converged and define the attainment progress as $\frac{current\ epoch}{convergence\text{-}line}$. For the jobs that failed to converge, we use $\frac{current\ epoch}{max\ epochs}$ instead.
- *Runtime-oriented attainment progress*: The runtime-oriented attainment progress is denoted as $\frac{current\ epoch}{completion\ criteria}$, which is further exemplified by the following case. If a job has a runtime-oriented completion criterion of 15 epochs, and the attainment progress is $\frac{5}{15} = 33.3\%$ after training 5 epochs.

We evaluate the Rotary-DLT against three baselines:

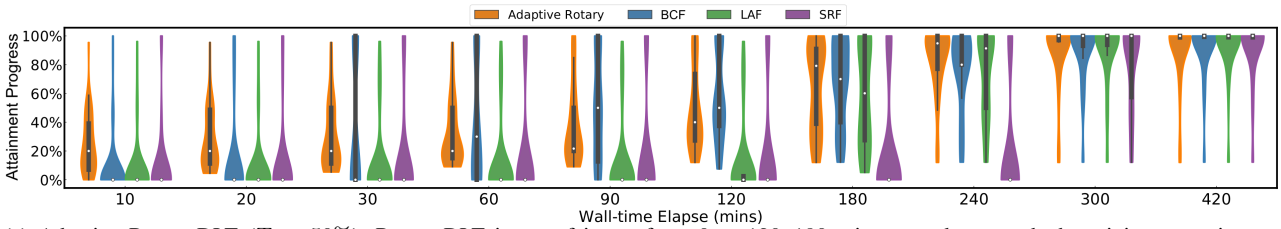
(a) Shortest Runtime First (SRF): it always runs the jobs with the shortest runtime completion criteria first and handles the other jobs following a round-robin strategy.
(b) Biggest Convergence First (BCF): it always runs the jobs with the biggest convergence completion criteria first and handles the other jobs following a round-robin strategy.
(c) Lowest Accuracy First (LAF): it always runs the jobs with the lowest accuracy completion criteria first and handles the other jobs following a round-robin strategy.

We demonstrate all the results in Fig. 10 using violin plots. In Fig. 10a, Rotary-DLT is adaptive, which fuses the fairness and efficiency policy. It starts with the pure-fairness policy that always selects the jobs with the lowest ϕ. Once all the jobs in the workload either achieve at least 50% progress toward their completion criteria or are considered converged, adaptive Rotary-DLT switches to an efficiency-centric policy, which starts to pick up the jobs with the highest ϕ. Fig. 10b and 10c demonstrate the performance of two Rotary-DLT variants that optimize fairness and efficiency objectives, respectively. Rotary-DLT variants outperform the three baselines. The threshold T is predefined in the evaluation to show the performance of Rotary under different scenarios; designing a sophisticated mechanism to choose T is out of the scope of this paper.

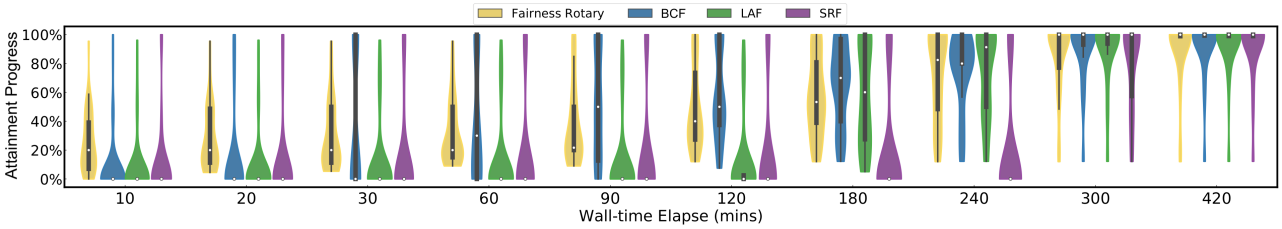
*3) **Impact of Training Epoch Estimation**:* Training epoch estimation is positioned at a vital place in developing and

Model	Architecture	Inception [53], MobileNet [54], MobileNetV2 [55], SqueezeNet [56], ShuffleNet [57], ShuffleNetV2 [58], ResNet [41], ResNeXt [59], EfficientNet [60], LeNet [61], VGG [62], AlexNet [63], ZFNet [64], DenseNet [65], LSTM [66], Bi-LSTM [42], BERT [67]
	Batch size	Computer vision models: 2, 4, 8, 16, 32 [51] Natural language processing models: 32, 64, 128, 256
	Optimizer	SGD, Adam, Adagrad, Momentum
	Learning rate	0.1, 0.01, 0.001, 0.0001, 0.00001
	Dataset	Computer vision models: CIFAR-10 [68] Natural language processing models: UD Treebank [69], Large Movie Review Dataset [70]
Completion Criteria	Convergence-oriented criteria (delta accuracy)	5%, 3%, 1%, 0.5%, 0.3%, 0.1%, 0.05%, 0.03%, 0.01%, 0.005%, 0.003%, 0.001%
	Accuracy-oriented criteria (final accuracy)	70%, 72%, 74%, 76%, 78%, 80%, 82%, 84%, 86%, 88%, 90%, 92%
	Runtime-oriented criteria (epoch)	From scratch 5, 10, 30, 50, 100 Pre-trained (Fine-tuned): 1, 2, 3, 4, 5
	Maximum epoch for criteria	1, 5, 10, 15, 20, 25, 30
Workload	Synthetic workload	60% DLT jobs with convergence-oriented completion criteria 20% DLT jobs with accuracy-oriented completion criteria 20% DLT jobs with runtime-oriented completion criteria

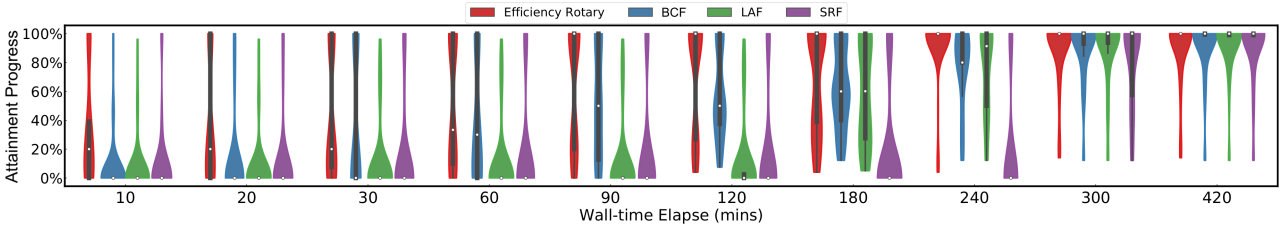
TABLE II: Synthetic DLT workload. The selection of model architecture and proportion of jobs with various completion criteria distribution are based on the responses to our survey, and the selection of other hyperparameters and the parameters about completion criteria follow the uniform distribution.



(a) Adaptive Rotary-DLT ($T = 50\%$): Rotary-DLT is pure-fairness from 0 to 120~180 minutes and can push the minimum attainment progress of the workload. Rotary-DLT becomes more aggressive on efficiency and completes more jobs starting from 180~240 minutes since all the jobs either make substantial attainment progress (50%) or are considered converged.



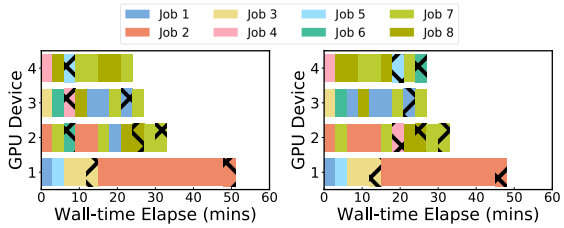
(b) Fairness Rotary-DLT ($T = 100\%$): Rotary-DLT always picks up the jobs with the lowest ϕ and can maximize the minimum attainment progress of all jobs in the workloads considerably faster than other baselines. For example, Fairness Rotary-DLT and SRF achieve the same minimum attainment progress for all jobs using 120 minutes and 300 minutes.



(c) Efficiency Rotary-DLT ($T = 0\%$): Rotary-DLT always selects the jobs with the highest ϕ and makes more jobs meet their completion criteria (achieving a higher attainment rate) in a relatively short period. Considering the results at 120 minutes, Efficiency Rotary-DLT completes more jobs than the other baselines.

Fig. 10: Evaluation of Rotary-DLT variants and three baselines (BCF, LAF, SRF) on the synthetic DLT workload

evaluating Rotary, and it is critical to understand its effect of it. We conduct a micro-benchmark workload with 8 DLT jobs and track the job placement under efficiency Rotary-DLT with and without accurate epoch estimation. Among eight jobs, $job4$ is for BERT, job 5 is for Bi-LSTM, and job 6 is for LSTM. To evaluate how the epoch estimation impacts the performance, we remove all the historical jobs about NLP models in the repository of Rotary-DLT so that the estimation for jobs 4,



(a) With Reliable Estimation (b) With Erroneous Estimation

Fig. 11: Job placements under efficiency Rotary-DLT.

5, and 6 are unreliable and even erroneous (e.g., the number of epochs for meeting the completion criteria is 2, but an erroneous estimate can be 100 epochs).

We demonstrate the placements for eight jobs in Fig. 11. Each rectangle denotes a job placement, and the one with hatches means the job meets the completion criteria. Fig. 11a presents the job placement under efficiency Rotary-DLT with the accurate epoch estimation. In light of the accurate epoch estimate, jobs $4, 5$, and 6 are triggered to run after the trial phase in Rotary-DLT and complete early. However, as shown in Fig. 11b, the epoch estimate is inaccurate, and the placement is inefficient accordingly. For example, job 4 can reach the complete criteria in 2 epochs, but the inaccurate estimate for that is 125 epochs, so its progress ϕ is much lower than others and cannot be placed as it should be. Therefore, jobs 4, 5, and 6 are finished later than those under accurate estimation.

*4) **Overhead of TTR, TEE, and TME**:* We investigate the overhead of recording the training epoch time of DLT jobs, namely measuring how the overhead of TTR and TEE in Rotary-DLT scales when the DLT workload grows. As shown in Table III, taking the workloads with the sizes of 10, 20, 30, and 40 as examples, the overhead of TTR and TEE takes an imperceptible proportion of the whole workload processing time, even for the larger workload.

Workload Size	Overall Running Time	Overhead of TTR	Overhead of TEE	Overhead of TME
10	8142s	0.225s	0.74s	0.58s
20	23790s	0.6s	1.31s	1.03s
30	34014s	0.87s	1.98s	1.49s
40	43124s	1.12s	2.56s	2.11s

TABLE III: The overall process time and overhead in Rotary

VI. Discussion

We discuss implementation choices and open questions in this section.

Implementation Choices: We faced several design trade-offs when implementing Rotary-AQP and Rotary-DLT. However, it should be noted that all the trade-offs are implementation-specific and framework independent, which could be mitigated by different implementations. We discuss two examples.

One implementation trade-off is how to persist the AQP jobs that have been paused (i.e., deferred to future execution) due to resource arbitration. When a job is paused, its intermediate states and results should be persisted either in memory or disk so that it can be resumed. Persisting AQP jobs in memory is more efficient from the perspective of performance but may quickly saturate the memory, which is a relatively scarce resource compared with disk and may lead to an out-of-memory error. Therefore, we checkpoint the AQP jobs in disks. Such a mechanism will bring additional overhead but allows more jobs to run simultaneously. The same issue happened when we implemented Rotary-DLT; however, checkpointing DLT jobs in disks is a common practice.

Our second implementation choice assumes the AQP and DLT jobs are executed in a single machine, even though our framework and system implementations support distributed execution. This is because we decide to first make a deep investigation of a resource arbitration framework and its implementations so that we can have a better understanding of progressive iterative analytic jobs and verify our framework design. Our system implementations, Rotary-AQP and Rotary-DLT, and the corresponding evaluations confirm the generality and practicality of the proposed framework. Thus, processing distributed jobs is out of the scope of this paper.

Materialization for Progressive Iterative Analytic: Progressive iterative analytic jobs need to be persisted. Such a requirement essentially asks for a materialization mechanism as in database systems and brings a similar trade-off between cost and efficiency [71]. How and when to materialize the progressive iterative analytic jobs is an interesting and pivotal research question, and we leave the answers for future work.

Unified Resource Arbitration Framework: While we compare AQP and DLT and treat them as two alike progressive iterative analytic applications in different areas and implement two systems for both of them, it is more interesting to have a unified resource arbitration system on a cluster to handle AQP and DLT jobs together. Such a system can serve more users and enormously improve resource utilization.

VII. Conclusion

This paper argues that resource arbitration is vital but neglected for progressive iterative analytic applications. We proposed a framework, Rotary, to highlight the core features and components for resource arbitration. It allows diverse user-defined completion criteria, prioritizes the jobs for resources, and supports adaptive running epochs. To realize and verify the framework, we implement two resource arbitration systems for AQP and DLT and evaluate them using the TPC-H benchmark and a survey-based workload, respectively. The evaluation results show that Rotary-AQP and Rotary-DLT outperform the state-of-the-art and other widely-used baselines and confirm that Rotary is an appealing solution for efficient resource utilization for iterative applications. Our work also opens interesting opportunities to explore the connection between research problems in ML and DB, such as balancing accuracy and running time in approximate query processing and deep learning training.

Acknowledgment

This research was supported in part by NSF Award IIS-2048088 and a Google DAPA Award. We would like to thank the anonymous reviewers for their insightful feedback.

REFERENCES

[1] S. Krishnan, A. J. Elmore, M. J. Franklin, J. Paparrizos, Z. Shang, A. Dziedzic, and R. Liu, "Artificial intelligence in resource-constrained and shared environments," *ACM SIGOPS Operating Systems Review*, vol. 53, no. 1, pp. 1–6, 2019.

[2] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for modern deep learning research," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2020, pp. 13 693–13 696.

[3] A. Crotty, A. Galakatos, C. Luckett, and U. Çetintemel, "The case for in-memory OLAP on "wimpy" nodes," in *IEEE International Conference on Data Engineering (ICDE)*, 2021, pp. 732–743.

[4] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin, "Crocodiledb: Efficient database execution through intelligent deferment," in *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[5] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Godwal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, J. Leeka, K. Park, H. Patel, O. Poppe, F. Psallidas, R. Ramakrishnan, A. Roy, K. Saur, R. Sen, M. Weimer, T. Wright, and Y. Zhu, "Cloudy with high chance of DBMS: a 10-year prediction for enterprise-grade ML," in *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[6] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[7] A. Agiwal, K. Lai, G. N. B. Manoharan, I. Roy, J. Sankaranarayanan, H. Zhang, T. Zou, J. Chen, M. Chen, M. Dai, T. Do, H. Gao, H. Geng, R. Grover, B. Huang, Y. Huang, A. Li, J. Liang, T. Lin, L. Liu, Y. Liu, X. Mao, M. Meng, P. Mishra, J. Patel, R. Sr, V. Raman, S. Roy, M. S. Shishodia, T. Sun, J. Tang, J. Tatemura, S. Trehan, R. Vadali, P. Venkatasubramanian, J. Zhang, K. Zhang, Y. Zhang, Z. Zhuang, G. Graefe, D. Agrawal, J. F. Naughton, S. Kosalge, and H. Hacigümüs, "Napa: Powering scalable data warehousing with robust query performance at google," *VLDB Endowment*, vol. 14, no. 12, pp. 2986–2998, 2021.

[8] Y. Fu and C. Soman, "Real-time data infrastructure at uber," in *ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 2503–2516.

[9] L. A. Melgar, D. Dao, S. Gan, N. M. Gürel, N. Hollenstein, J. Jiang, B. Karlaš, T. Lemmin, T. Li, Y. Li, S. Rao, J. Rausch, C. Renggli, L. Rimanic, M. Weber, S. Zhang, Z. Zhao, K. Schawinski, W. Wu, and C. Zhang, "Ease.ml: A lifecycle management system for mldev and mlops," in *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop YARN: yet another resource negotiator," in *ACM Symposium on Cloud Computing (SOCC)*, 2013, pp. 5:1–5:16.

[11] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 2:1–2:14.

[12] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *European Conference on Computer Systems (EuroSys)*, 2016, pp. 35:1–35:16.

[13] H. Herodotou and E. Kakoulli, "Trident: Task scheduling over tiered storage systems in big data platforms," *VLDB Endowment*, vol. 14, no. 9, pp. 1570–1582, 2021.

[14] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *European Conference on Computer Systems (EuroSys)*, 2018, pp. 3:1–3:14.

[15] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 595–610.

[16] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 485–500.

[17] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 947–960.

[18] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. M. Lau, Y. Wang, Y. Xiong, and B. Wang, "Hived: Sharing a GPU cluster for deep learning with guarantees," in *USENIX Symposium on Operating Systems Design and Implementation OSDI*, 2020, pp. 515–532.

[19] B. Wagner, A. Kohn, and T. Neumann, "Self-tuning query scheduling for analytical workloads," in *ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 1879–1891.

[20] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[21] C. Lyu, Q. Fan, F. Song, A. Sinha, Y. Diao, W. Chen, L. Ma, Y. Feng, Y. Li, K. Zeng, and J. Zhou, "Fine-grained modeling and optimization for intelligent resource management in big data processing," *VLDB Endowment*, vol. 15, no. 11, p. 3098–3111, 2022.

[22] R. Liu, D. Wong, D. Lange, P. Larsson, V. Jethava, and Q. Zheng, "Accelerating container-based deep learning hyperparameter optimization workloads," in *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, 2022.

[23] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, pp. 185:1–185:52, 2017.

[24] "Apache Spark," https://spark.apache.org, 2022, accessed: 2022-10-08.

[25] "TensorFlow," https://www.tensorflow.org, 2022, accessed: 2022-10-08.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *VLDB Endowment*, vol. 3, no. 1, pp. 285–296, 2010.

[27] S. Floratos, Y. Zhang, Y. Yuan, R. Lee, and X. Zhang, "Sqloop: High performance iterative processing in data management," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1039–1051.

[28] K. Li and G. Li, "Approximate query processing: What is new and where to go? - A survey on approximate query processing," *Data Science and Engineering*, vol. 3, no. 4, pp. 379–397, 2018.

[29] S. Chaudhuri, B. Ding, and S. Kandula, "Approximate query processing: No silver bullet," in *ACM International Conference on Manageme of Data (SIGMOD)*, 2017, pp. 511–519.

[30] K. Zeng, S. Agarwal, and I. Stoica, "iolap: Managing uncertainty for efficient incremental OLAP," in *International Conference on Management of Data (SIGMOD)*, 2016, pp. 1347–1361.

[31] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *International Conference on Management of Data (SIGMOD)*, 2014, pp. 481–492.

[32] L. Stafman, A. Or, and M. J. Freedman, "Relaqs: Reducing latency for multi-tenant approximate queries via scheduling," in *International Middleware Conference (Middleware)*, 2019, pp. 280–292.

[33] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 1049–1062.

[34] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for DNN training," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 1–15.

[35] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on GPU clusters for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 533–548.

[36] B. Andersson, S. K. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2001, pp. 193–202.

[37] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[38] R. I. Davis and A. J. Wellings, "Dual priority scheduling," in *IEEE Real-Time Systems Symposium (RTSS)*, 1995, pp. 100–109.

[39] X. Yang and N. H. Vaidya, "Priority scheduling in wireless ad hoc networks," in *ACM Interational Symposium on Mobile Ad Ho Networking and Computing (MobiHoc)*, 2002, pp. 71–79.
[40] D. Alistarh, J. Kopinsky, J. Li, and G. Nadiradze, "The power of choice in priority scheduling," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017, pp. 283–292.
[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
[42] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, 2005.
[43] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.
[44] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *European Conference on Computer Systems (EuroSys)*, 2013, pp. 29–42.
[45] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *ACM International Conference on Management of Data (SIGMOD)*, 1997, pp. 171–182.
[46] S. Chaudhuri, G. Das, and V. R. Narasayya, "Optimized stratified sampling for approximate query processing," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 2, p. 9, 2007.
[47] S. Kay, *Fundamentals of statistical signal processing*. Prentice Hall PTR, 1993.
[48] "Cost-based optimizer," https://docs.databricks.com/spark/latest/spark-sql/cbo.html, 2022, accessed: 2022-10-08.
[49] R. Liu, S. Krishnan, A. J. Elmore, and M. J. Franklin, "Understanding and optimizing packed neural network training for hyper-parameter tuning," in *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, 2021, pp. 3:1–3:11.
[50] "Apache kafka," https://kafka.apache.org, 2022, accessed: 2022-10-08.
[51] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *CoRR*, vol. abs/1804.07612, 2018. [Online]. Available: http://arxiv.org/abs/1804.07612
[52] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, vol. 7700, pp. 437–478.
[53] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.
[54] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.
[55] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.
[56] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.
[57] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856.
[58] N. Ma, X. Zhang, H. Zheng, and J. Sun, "Shufflenet V2: practical guidelines for efficient CNN architecture design," in *European Conference on Computer Vision (ECCV)*, vol. 11218, 2018, pp. 122–138.
[59] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5987–5995.
[60] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning (ICML)*, vol. 97, 2019, pp. 6105–6114.
[61] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceeding of IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[62] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.
[63] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2012, pp. 1106–1114.
[64] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European Conference on Computer Vision (ECCV)*, vol. 8689, 2014, pp. 818–833.
[65] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.
[66] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[67] I. Turc, M. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: The impact of student initialization on knowledge distillation," *CoRR*, vol. abs/1908.08962, 2019.
[68] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," 2009.
[69] "Universal dependencies," https://universaldependencies.org, 2022, accessed: 2022-10-08.
[70] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Annual Meeting of the Association for Computational Linguistics ACL*, 2011, pp. 142–150.
[71] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *International Conference on Data Engineering (ICDE)*, 2007, pp. 466–475.