THE UNIVERSITY OF CHICAGO

RESOURCE-AWARE OPTIMIZATIONS FOR DATA-INTENSIVE SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
RUI LIU

CHICAGO, ILLINOIS
DECEMBER 2023

Copyright © 2023 by Rui Liu

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vii

ACKNOWLEDGMENTS viii

ABSTRACT x

1 INTRODUCTION 1
- 1.1 Primitive 1: Resource Utilization Maximization 3
- 1.2 Primitive 2: Resource Arbitration 3
- 1.3 Primitive 3: Resource Suspension and Resumption 5
- 1.4 Contribution 7

2 REPACK: UNDERSTANDING AND OPTIMIZING REPACKED NEURAL NETWORK TRAINING FOR HYPER-PARAMETER TUNING 10
- 2.1 Background 12
  - 2.1.1 Motivation 12
  - 2.1.2 Basic Framework API 13
- 2.2 Implementation 15
  - 2.2.1 Packing 15
  - 2.2.2 Misaligned Batch Sizes 16
  - 2.2.3 Misaligned Step Counts 17
  - 2.2.4 Eliminating Redundancy 17
- 2.3 Profiling Model Packing 19
  - 2.3.1 Profiling Setup 19
  - 2.3.2 Profiling Metrics 19
  - 2.3.3 Improvement 21
  - 2.3.4 Memory Usage 22
  - 2.3.5 Switching Overheads 23
  - 2.3.6 `Pack` vs CUDA Parallelism 23
  - 2.3.7 Ablation Study 24
- 2.4 `Pack`-Aware Hyperparameter Tuning 26
  - 2.4.1 Hyperband 27
  - 2.4.2 `Pack`-aware Hyperband 28
  - 2.4.3 Evaluation for Hyperparameter Tuning 29
- 2.5 Related Work 31
  - 2.5.1 Systems for Hyperparameter Tuning 31
  - 2.5.2 Systems for Multi-tenancy 32
- 2.6 Discussion 33

3 ROTARY: A RESOURCE ARBITRATION FRAMEWORK FOR PROGRESSIVE ITERATIVE ANALYTICS . . . 35
3.1 Related Work . . . 39
3.1.1 Scheduling for AQP . . . 39
3.1.2 Scheduling for Machine Learning . . . 40
3.1.3 Multi-tenant Systems . . . 40
3.2 Resource Arbitration Framework . . . 41
3.2.1 Terminology and Setup . . . 41
3.2.2 User-defined Completion Criteria . . . 42
3.2.3 Framework Architecture . . . 44
3.2.4 Resource Arbitration Problem Statement . . . 46
3.2.5 Resource Arbitration Algorithm . . . 47
3.3 System Implementation . . . 48
3.3.1 Rotary-AQP Implementation . . . 48
3.3.2 Rotary-DLT Implementation . . . 51
3.4 Evaluation . . . 54
3.4.1 Rotary-AQP Evaluation . . . 55
3.4.2 Rotary-DLT Evaluation . . . 60
3.5 Discussion . . . 65

4 RIVETER: ADAPTIVE QUERY SUSPENSION AND RESUMPTION FRAMEWORK FOR CLOUD NATIVE DATABASES . . . 67
4.1 Motivation . . . 69
4.1.1 Query Suspension and Resumption . . . 69
4.1.2 Motivational Cases . . . 71
4.2 Riveter Design and Implementation . . . 72
4.2.1 Pipeline-level Suspension and Resumption . . . 73
4.2.2 Process-level Suspension and Resumption . . . 74
4.2.3 Resource-adaptive Query Execution . . . 76
4.3 Evaluation . . . 81
4.3.1 Impact of Intermediate Data Persistence during Suspension . . . 82
4.3.2 Analysis of Suspension and Resumption Strategy Selection . . . 87
4.3.3 Accuracy Estimation and Runtime of Cost Model . . . 90
4.4 Related Work . . . 92
4.4.1 Database Migration . . . 92
4.4.2 Recovery, Checkpoint, and Suspension . . . 93
4.4.3 Query Scheduling . . . 95
4.5 Discussion . . . 96

5 CONCLUSION . . . 98

REFERENCES . . . 100

LIST OF FIGURES

1.1 Iterative and progressive queries 5

2.1 The dataflow of a training step in the single mode v.s. the packed mode. The training data resides in main memory and is copied over to the device in batches during each training step, resulting in a backpropagation computation and then a parameter update. By synchronizing the dataflow, the packed mode can reuse work when possible. 13
2.2 All the models share the batch input stream, each batch is padded and sliced for training the packed model. 16
2.3 Early finished model is freed and checkpointed, and a new model is packed with the others for further training. 18
2.4 *Improvement* of packing models when increasing the number of models and batch size on GPU. The Y-axis indicates the reduction in training time compared to sequential execution (as defined in Equation 2.3). 22
2.5 GPU memory peak of different models 22
2.6 $T_s(Seq)$ vs. $T_s(Pack)$ (milliseconds) when packing two models on a GPU 26

3.1 Progress curves of AQP and DLT jobs 37
3.2 Work Positioning 38
3.3 Templates of user-defined completion criteria 43
3.4 Examples of user-defined completion criteria 43
3.5 Framework architecture of Rotary 45
3.6 Evaluation of Rotary-AQP and four baselines (Round-robin, EDF, LAF, ReLAQS) on the synthetic AQP workload 56
3.7 False attainment and waiting time of Rotary-AQP 58
3.8 Attained jobs in the various workloads (30 jobs) 59
3.9 Impact of progress estimation 60
3.10 Evaluation of Rotary-DLT variants and three baselines on the synthetic DLT workload . . 63
3.11 Job placements under efficiency Rotary-DLT. 64

4.1 Riveter architecture 73
4.2 Workflow of pipeline-level query suspension and resumption strategy 73
4.3 Pipeline-level suspension and resumption strategy for common operators in Riveter 75
4.4 Workflow of process-level query suspension and resumption 76
4.5 Illustrative example of query suspension and resumption algorithm 81
4.6 Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the process-level strategy on SF-10, SF-50, SF-100 datasets 83
4.7 Persisted intermediate data size of queries (Q1, Q3, Q17, Q21) in TPC-H when suspending them at around 30%, 60%, and 90% execution time using process-level strategy on SF-100 dataset 84
4.8 Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the pipeline-level strategy on SF-10, SF-50, SF-100 datasets 85
4.9 Time lag incurred when suspensions are requested under the pipeline-level strategy 86
4.10 Analysis of strategy selection for Q1 87

4.11 Analysis of strategy selection for Q3 88
4.12 Analysis of strategy selection for Q17 89
4.13 Analysis of strategy selection for Q21 90

LIST OF TABLES

1.1 Analysis of suspension & resumption strategies 7

2.1 $SwOH$ of training two models sequentially 23
2.2 Model configurations for ablation study 25
2.3 Hyperparameter configurations for evaluation 30
2.4 Performance of `pack`-aware Hyperband 31

3.1 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. 57
3.2 Synthetic DLT workload. The selection of model architecture and proportion of jobs with various completion criteria distribution is based on the responses to our survey, and the selection of other hyperparameters and the parameters about completion criteria follow the uniform distribution. 61
3.3 The overall process time and overhead in Rotary 65

4.1 Representative suspension & resumption strategies 71
4.2 Selected queries in TPC-H 82
4.3 Estimation analysis of cost model when queries are suspended at around 50% using process-level strategy 91
4.4 Running time of cost model in Riveter based on the queries using SF-100 dataset 91

ACKNOWLEDGMENTS

Words cannot truly convey my appreciation, but to everyone in my Ph.D. journey, this is for you:

I first want to express my deepest gratitude to Prof. Aaron J. Elmore. I have not only learned the intricacies of becoming a proficient and independent researcher but have also developed a research-oriented mindset. Your methodical guidance and invaluable insights consistently illuminated my path whenever I encountered challenges in my academic journey. In retrospect, I still marvel at how I managed to pass your Ph.D. interview, as I struggled to grasp your questions at that time. Fast forward six years, and I find myself composing this dissertation, a testament to the remarkable transformation I have undergone as your Ph.D. student.

Prof. Michael J. Franklin, I still felt audacious when I approached you to be my Ph.D. advisor, and I couldn't be more grateful for that initial bold step. Throughout my Ph.D. studying, your guidance has been nothing short of exceptional, consistently providing me with high-level and insightful advice. I vividly remember your counsel about the importance of aspiring to be known as an author or inventor of something truly groundbreaking rather than merely being recognized for authoring a multitude of papers by the end of my Ph.D. program. Your words have lingered in my thoughts, prompting deep reflection on the path I have chosen.

Prof. Sanjay Krishnan, I have always regarded you as my "unofficial" third advisor, given the wealth of insightful suggestions and guidance you've generously provided. Your research insights have played a pivotal role in kindling my passion for machine learning and artificial intelligence. I am genuinely awed by the multitude of research ideas you possess and the unwavering rigor in your approach. I often find myself aspiring to emulate your depth of research thinking. One unforgettable memory is how you transformed my initial paper into an entirely new creation. I am grateful that you maintained a consistent level of support throughout my Ph.D. journey but gradually introduced less extensive revision to my research paper, which, I believe, served as a confirmation of my Ph.D. progress.

Prof. Raul Castro Fernandez, among the faculty members in ChiData, you are the only one I didn't have the opportunity to collaborate with on a research project, or a better way to think of this is, you are

the only faculty member I didn't find a chance to bother. Feel free to ask Aaron, Mike, and Sanjay, and you will know how lucky you are. Nevertheless, our interactions weren't solely professional; we shared moments over drinks, and I often invited you to join me, sometimes even for a bottom-up drinking. While I sense that this might not have been your favorite pastime, I want to extend my sincere apologies if I may have overdone it, even if I probably will keep doing this in the future.

My academic siblings in ChiData group, I treasure our joyous moments and connections: Chunwei Liu, Xi Liang, Hao Jiang, Ted Shao, Jinjin Zhao, Will Brackenbury, Bruno Barbarioli, Gabe Mersy, Dixin Tang, Stavros Sintos, Suhail Rehman, Jun Hyuk Chang, Zhe Heng Eng, Riki Otaki, Yue Gong, Zhiru Zhu, Chris Zhu, Tapan Srivastava, Zechao Shang, Steven Xia. For the past, current, and future members whom I cannot enumerate, you are not strangers to me; as long as we all have the ChiData brand, you are my friends.

Moyao Liu, Wenbo Tang, Guanglu Xue, Jiaxin Sun, Rongtai Lu, Song Xue, Xiao Liu, Tao Li, my dear friends, your every presence brightens my days.

A special thanks to my wife, Tengteng Ma, who makes me home when I escape from research. I am proud that you got a Ph.D. degree sooner than me.

Mother and Father, your endless love and care throughout my life. No matter how far away, you are always my biggest and best supporters. I love you and miss you, though I never say it.

ABSTRACT

In modern cloud computing environments, characterized as resource-dynamic with new developments, ephemeral cloud resources are becoming increasingly prevalent. Ephemeral resources exhibit two distinct characteristics: (1) they can be terminated either predictably or unexpectedly by resource providers during utilization, and (2) their prices, while often more cost-effective than reserved or on-demand cloud resources, can fluctuate over time. Deploying data-intensive systems on ephemeral cloud resources to process jobs for multi-tenants presents two challenges. First, the availability of resources can be significantly lower than the number of jobs when resources are inaccessible due to their ephemeral nature or rendered impractical due to exorbitant price fluctuations. Second, when resources are terminated or experience unreasonably high price fluctuations, the necessity to terminate jobs results in the forfeiture of ongoing progress. To address the aforementioned challenges, we present three optimizations: maximizing resource utilization for data-intensive jobs when resources are available, preempting and reallocating scarce resources to the most appropriate jobs, and suspending the jobs when necessary or advantageous and resuming them later to skip resource termination or unavailability without losing substantial progress. In this dissertation, we develop various prototype systems to realize the optimizations.

Firstly, we propose and implement Repack for deep learning training to share common I/O and computing processes among models on the same computing device. We further present a comprehensive empirical study and end-to-end experiments that suggest significant improvements for hyperparameter tuning. The results suggest: (1) packing two models can bring up to 40% performance improvement over unpacked setups for a single training step, and the improvement increases when packing more models; (2) the benefit of the pack primitive largely depends on a number of factors including memory capacity, chip architecture, neural network structure, and batch size; (3) there exists a trade-off between packing and unpacking when training multiple neural network models on limited resources; (4) a `pack`-aware Hyperband is up to 2.7× faster than the original Hyperband, with this improvement growing as memory size increases and subsequently the density of models packed.

Secondly, we propose and design a resource arbitration framework, Rotary, to continuously priori-

tize the progressive iterative analytics and determine if/when to reallocate and preempt the resources for them. 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. Within Rotary, we consider two prevalent cases: approximate query processing (AQP) and deep learning training (DLT). Based on Rotary, we 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. 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.

Finally, we present an adaptive query suspension and resumption framework, Riveter, for deploying cloud-native databases in scenarios where resource availability becomes ephemeral and resource monetary costs fluctuate. Riveter can suspend queries when the resources are limited, or the costs are unexpectedly high, then resume them when the resources become available and cost-effective. Within Riveter, we implement various strategies, including (1) a redo strategy that terminates queries and subsequently re-runs them, (2) a pipeline-level strategy that suspends a query once one of its pipelines has completed, thereby reducing the storage requirements for intermediate data, (3) and a process-level strategy that enables the suspension of query execution processes at any given moment but generates a substantial volume of intermediate data for query resumption. We also devise a cost model to determine the query suspension and resumption strategy that causes minimum latency for various queries. To demonstrate the effectiveness of Riveter, we conducted a performance study, an end-to-end analysis, and a cost model evaluation using the TPC-H benchmark. Our results present the difference among the suspension and resumption strategies of Riveter in terms of the size of persisted intermediate data and confirm the adaptive and efficient query suspension and resumption delivered by Riveter.

CHAPTER 1

INTRODUCTION

Data-intensive systems are migrating towards cloud-native architectures that offer low-latency, consistent, and pay-as-you-go query answering [85]. This is not just a deployment trend but an important point to reconsider the core architectural decisions that underpin these systems.

In modern cloud environments, resources that are dynamic in availability and monetary cost are becoming prevalent. First, there is an increase in ephemeral cloud resources. Spot instances, offering short-lived computing infrastructure, have been prevalent for a while [6, 177, 29]. However, new developments are amplifying the bursty capacity. For instance, serverless platforms offer applications the convenience of employing lightweight Virtual Machines (VMs) that have limited run-time, memory capacity, and addressable addresses [67]. Recent efforts from the database community have shown how to develop a query processing framework on top of a serverless platform [126]. Another emerging cloud paradigm is "zero-carbon clouds" [28], where data centers can be completely ephemeral as they are largely powered by renewable sources that fluctuate. Second, the prices of cloud resources can be significantly dynamic as well. Their prices during peak demand can reportedly skyrocket to 200 to 400 times the normal rate [140]. Thus, although users expect low latency, there is an increased demand for economically viable solutions, provided they do not significantly compromise performance. A growing number of users are beginning to favor cost-conscious options that may result in slightly increased latency or stale results [4]. Such dynamic resources pose a new opportunity for systems designers to reduce the operational costs of database systems. However, the ephemeral and fluctuating nature of such resources is often incompatible with current database designs and workloads. The convention of data-intensive systems pre-reserving what are assumed to be stable cloud resources to maintain low latency has become less applicable in these resource-dynamic environments. Deploying data-intensive systems on ephemeral and fluctuating cloud resources poses two distinct challenges from the perspective of cloud resources managers or providers:

1. The availability of resources can be significantly lower than the number of jobs when resources are

inaccessible due to their ephemeral nature or rendered impractical due to exorbitant price fluctuations.

2. When resources are terminated or experience unreasonably high price fluctuations, the necessity to terminate jobs results in the forfeiture of ongoing progress.

We believe that a novel data-intensive system is necessary to address the challenges. This requires rethinking the design principles of data-intensive systems and necessitates the following primitives:

Primitive 1: Resource Utilization Maximization. In light of the ephemeral and fluctuating nature of resources in availability and costs, it is imperative to optimize resource utilization during periods when they are accessible or economically feasible to employ. Consequently, a fine-grained resource-sharing mechanism becomes vital. This mechanism not only facilitates resource sharing and allocation among users for short-term purposes but also ensures efficient resource utilization.

Primitive 2: Resource Arbitration. Given the ephemeral and fluctuating resources, "how many resources does a query need?", a question answered by existing resource reservation approaches, no longer holds substantial value. Instead, the emergent question prompted by this primitive is: "Is it worth allocating resources to a particular query?". Answering this question necessitates a sophisticated mechanism that can adaptively determine if, when, how much, and for how long a query should be allocated resources. This decision should consider multiple factors, such as user needs, available resources, and the progress of each query at various times.

Primitive 3: Resource Suspension and Resumption. The feasibility of permitting queries to utilize ephemeral cloud resources constantly is limited because (1) the resources are dynamic in availability and cost, making long-term reservation and sustainability unfeasible, (2) the continuous allocation of constrained resources to long-running queries, particularly when they yield diminishing returns, can be counterproductive. Thus, such queries ought to be paused when necessary or beneficial. This primitive fosters more efficient and flexible query execution by transforming a single long-running query into a sequence of shorter ones.

1.1 Primitive 1: Resource Utilization Maximization

A data-intensive system focused on maximizing resource utilization can enhance the granularity of resource sharing and utilization for users. However, previous data-intensive systems have been inefficient in terms of resource sharing. One of the central issues contributing to this inefficiency is the fact that their design and implementation often overlook the specific characteristics of resources and workloads. These systems tend to treat modern data-intensive jobs much like traditional data processing jobs, failing to recognize the unique demands of the resources and jobs at hand. For instance, deep learning training jobs heavily rely on specialized hardware, such as GPUs. For example, deep learning training jobs rely on specialized hardware such as GPUs, which makes fine-grained resource sharing (i.e., training multiple networks on the same device) significantly harder than the classic CPU-based computation job.

The ephemeral and fluctuating resources exacerbate the inefficiency of resource sharing. Since these resources are not consistently accessible, they often result in sub-optimal resource utilization. To achieve efficient resource sharing for data-intensive systems, we argue that it is necessary to leverage resource traits and exploit the characteristics of data-intensive jobs. For instance, consider the case of deep learning training jobs, which are typically executed in an iterative fashion. This iterative nature provides an opportunity to break down a long-running deep learning training job into multiple shorter, more manageable training tasks. Moreover, these training processes often occur on GPUs, where it is possible to accommodate and run multiple small, short-running jobs simultaneously for brief intervals. This fine-grained resource-sharing approach effectively addresses the challenge of optimizing resource utilization, which is particularly pertinent in the context of ephemeral and fluctuating resources.

1.2 Primitive 2: Resource Arbitration

Resource arbitration is a novel adaptive scheduling paradigm that can continuously reallocate and preempt resources. It is critical since resource preemption is only worthwhile if the benefits of reallocating the preempted resources exceed the overhead of preemption and the loss associated with the preempted data-intensive jobs, which is analogous to context switching. In comparison to classic scheduling

methods, it needs to consider various factors, such as cloud resources availability, characteristics of different queries (e.g., job completion criteria), the progress of each job, and specified users' needs. Specifically, with the recent proliferation of data-driven applications, many organizations have increasingly complex and heterogeneous workloads, including both long-running and short-running data-intensive jobs [100, 44]. Long-running jobs may occupy resources for extended periods and are not readily suited for dynamic resources. This can lead to significant delays for shorter-running jobs that might have otherwise been completed promptly with sufficient resources. Many of these long-running data-intensive jobs can often be progressive, where an iterative loop repeatedly refines a result until the desired completion criterion is met – these are the so-called iterative and progressive jobs [96]. To keep allocating limited and dynamic resources to jobs that have already achieved significant progress may, under certain conditions, curtail the efficient utilization of these resources.

Due to the ephemeral availability and fluctuating prices of cloud resources, resource arbitration for queries happens in an iterative way, resulting in the queries being processed iteratively either by design or inherently, as illustrated in Figure 1.1(a). Specifically, a novel data-intensive system can evaluate ongoing and preempted queries at each iteration and then reallocate the resources to the most promising queries, for example. the ones that can achieve the jobs quickly if the resources are granted. A novel data-intensive system that supports resource arbitration is also designed to handle complex and heterogeneous workloads. For instance, one feature of resource arbitration is the capability to manage queries that allow early termination (e.g., approximation) and demonstrate diminishing returns. As depicted in Figure 1.1(b), the job progress improvement between starting and ending points is not linear. Thus, in certain scenarios, it may be beneficial to suspend queries exhibiting diminishing returns and reallocate resources to queries that promise more significant progress in a shorter timeframe. However, in other scenarios, it is valuable to keep refining the results of data-intensive jobs by allocating sufficient resources continuously.

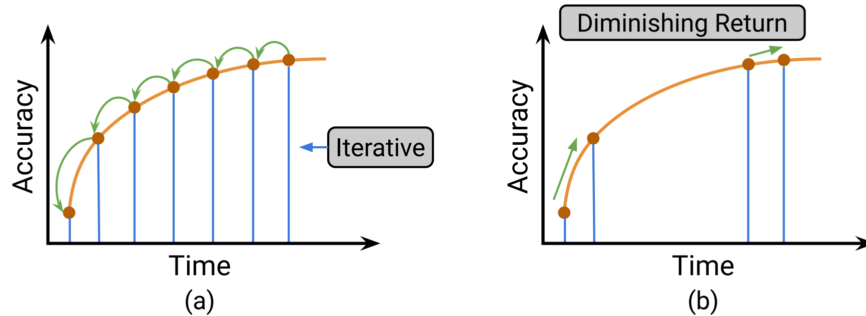


Figure 1.1: Iterative and progressive queries

1.3 Primitive 3: Resource Suspension and Resumption

Resource suspension and resumption entail the ability to pause a running data-intensive job in a partially complete state when the resources are not available or practical to exploit. The paused job can be resumed later when its continuation is considered feasible or beneficial. To achieve this primitive, we focus on a potential solution that holds promise: adaptive query suspension and resumption, a mechanism that can significantly enhance the flexibility and efficiency of cloud-native databases.

The original query suspension and resumption is proposed to create a query suspension plan for a pull-based model without updates, and minimize the overhead of serialization to disk during suspension [24]. This method presents certain constraints, such as both the suspended and resumed query require the same resources and database state, and a query plan is executed as a single thread. *We revisit this concept and believe that a novel solution is necessary to accommodate the new trends and realize the identified primitives in cloud-native databases.* Therefore, we take an initial step by designing a novel adaptive query suspension and resumption by integrating more strategies at various granularities. A vanilla strategy is to redo the query, which is rooted in recovery mechanisms [113, 106]. Another strategy performs at the level of data batch and places checkpoints between data batches during runtime, analogous to the established checkpoint mechanism in streaming-style systems [11]. These checkpoints serve as resumption points in cases of unexpected termination. Further advancements also ushered in suspension and resumption strategies during query execution. The original query suspension and resumption is an early instance performed at the operator level for pull-based execution [24]. There are two alternative strategies that

facilitate suspending and resuming queries at the pipeline and process levels. The pipeline-level strategy enables the query suspension and resumption for multi-threads pipeline-driven query execution and provides more flexibility (e.g., different resource configurations) when resuming queries. The process-level strategy operates within the context of a process, allowing a query within a process to be suspended and resumed at any given time by capturing and persisting all the intermediate data. We further detail these approaches later in this section.

Each aforementioned suspension and resumption strategy possesses distinct characteristics, and we proposed five metrics for a clear portrait from various perspectives:

- *Agility* is the speed at which the suspension can be initiated.
- *Capacity* measures the amount of intermediate data needed to be persisted during query suspension.
- *Adaptivity* designates whether the strategy can utilize available resources for query resumption.
- *Complexity* evaluates the development efforts to achieve such a strategy, e.g., is the data system modification needed.
- *Preservability* indicates the progress a strategy can preserve when a suspension or termination happens.

We conduct an analysis of different strategies in Table 1.1. For example, the redo strategy permits the termination of a query at any time without persisting any intermediate state for query resumption. It re-runs the query using currently available resources; thus, additional development or changes are unnecessary. Meanwhile, the process-level suspension and resumption strategy, typically relying on third-party tools, allows the suspension of queries at any given moment as well. This strategy preserves the current progress of the process as it keeps all intermediate data and context states. However, the practice of "storing everything" presents two significant downsides. First, the volume of persisted data can be exceedingly large. Second, it restricts the resumption of processes and queries to the same resource configurations (such as the number of hardware threads and the allocated memory size) that were in use at the time of suspension. The pipeline-level strategy offers a different approach by preserving the intermediate data

of each pipeline in the query plan for resumption, which implies that suspension and data persistence can only occur once certain pipelines have concluded. This approach can drastically reduce the volume of data that needs to be stored but only suspend queries at specific points. Furthermore, this strategy demands data systems alterations, as it modifies the way queries are executed.

	Agility	*Capacity*	*Adaptivity*	*Complexity*	*Preservability*
Redo	Terminate at anytime	No intermediate data & state	Redo queries with available resources	No additional efforts	Lost all progress
Pipeline-Level Suspend & Resume	Suspend until some pipeline complete	Persisted intermediate data & state of pipeline	Resumption with available resources	Modified data systems	Lost progress since last persisted pipeline
Process-Level Suspend & Resume	Suspend anytime at process level	Persisted intermediate data & state of process	Resumption needs same resources as suspension	Import additional toolkit	Preserved all progress

Table 1.1: Analysis of suspension & resumption strategies

Therefore, it is essential to adaptively suspend and resume queries at different granularities, particularly when dealing with complex and heterogeneous workloads in resource-dynamic cloud environments.

1.4 Contribution

We design and implement resource-aware optimizations for data-intensive systems and provide a high-level overview for each work as follows.

Repack: As neural networks are increasingly employed in machine learning practice, how to efficiently share limited training resources among a diverse set of model training tasks becomes a crucial issue. To achieve better utilization of the shared resources, we explore the idea of jointly training multiple neural network models on a single GPU in this paper. We realize this idea by proposing a primitive, called `pack`. We further present a comprehensive empirical study and end-to-end experiments that suggest significant improvements for hyperparameter tuning. The results suggest: (1) packing two models can bring up to 40% performance improvement over unpacked setups for a single training step, and the improvement

increases when packing more models; (2) the benefit of the `pack` primitive largely depends on a number of factors including memory capacity, chip architecture, neural network structure, and batch size; (3) there exists a trade-off between packing and unpacking when training multiple neural network models on limited resources; (4) a `pack`-aware Hyperband is up to 2.7× faster than the original Hyperband, with this improvement growing as memory size increases and subsequently the density of models packed.

Rotary: Increasingly modern computational applications employ progressive iterative analytics. In comparison to classic computation 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 diverse completion criteria, produce diminishing returns, and process data at a different rate, 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 for them. We propose and design a resource arbitration framework, Rotary, and we 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 heuristic baselines and confirm the generality and practicality of the proposed resource arbitration framework.

Riveter: In modern cloud environments, it is prevalent that resource availability and cost become ever-changing and resource fluctuating. Such a dynamic nature brings a novel challenge to the applications deployed in the cloud environments. For cloud-native databases, it is becoming increasingly significant to facilitate resource-adaptive query execution, namely suspending queries when the resources are limited, or the costs are unexpectedly high, then resuming them when the resources are available and acceptable to use. This novel and challenging task requires redesigning the classic query execution and devising a mechanism to adaptively determine if, when, and how to suspend queries. To address this

challenge, we present Riveter, a resource-adaptive query suspension and resumption framework that can pause ongoing queries and persist all the necessary intermediate states for potential query resumption in the future whenever such resumption is deemed feasible or beneficial. Within Riveter, we implement two query suspension and resumption strategies: (1) a pipeline-level strategy that suspends a query once one of its pipelines has completed, thereby reducing the storage requirements for intermediate states, (2) and a process-level strategy that enables the suspension of query execution processes at any given moment but generate a substantial volume of intermediate states for query resumption. We also devise a resource-oriented cost model in Riveter to determine query suspension and resumption. We evaluate Riveter based on the TCP-H benchmark, and the results show that Riveter can efficiently suspend and resume queries in scenarios where resources are ephemeral by minimizing the latency caused by suspension and resumption.

CHAPTER 2

REPACK: UNDERSTANDING AND OPTIMIZING REPACKED NEURAL NETWORK TRAINING FOR HYPER-PARAMETER TUNING

The successes of AI are in part due to the adoption of neural network models which can place immense demand on computing infrastructure. It is increasingly the case that a diverse set of model training tasks share limited training resources. The long-running nature of these tasks and the large variation in their size and complexity make efficient resource sharing a crucial concern. The concerns are compounded by an extensive trial-and-error development process where parameters are tuned and architectures have tweaked that result in a large number of trial models to train. Beyond the monetary and resource costs, there are long-term questions of economic and environmental sustainability [145, 137].

Efficiently sharing the same infrastructure among multiple training tasks, or multi-tenant training, is proposed to address the issue [168, 75, 120, 107]. The role of a multi-tenancy framework is to stipulate policies and constraints on how contended resources are partitioned and tasks are placed on physical hardware. Most existing approaches divide resources at the granularity of full devices (e.g., an entire GPU) [65]. Such a policy can result in low resource utilization due to its coarse granularity. For example, models may greatly vary in size, where the largest computer vision models require multiple GBs of GPU memory [20] but mobile-optimized networks use a significantly smaller space [135]. Given that GPUs today have significantly more on-board memory than in the past (e.g., up to 32 GB in commercial offerings), if a training workload consists of a large number of small neural networks, allocating entire devices to these training tasks is wasteful and significantly delays any large model training.

Furthermore, the reliance on specialized hardware such as GPUs makes fine-grained resource sharing (i.e., training multiple networks on the same device) significantly harder than the typical examples in cloud systems. Unlike CPUs, the full virtualization of GPU resources is nascent [124]. While modern GPU libraries support running multiple execution kernels in parallel, sharing resources using isolated kernels is not a mature solution in this setting. Many deep learning workloads are highly redundant, for example, the typical parameter tuning process trains the same model on the same data with small

tweaks in hyperparameters or network architectures. In this setting, those parallel kernels would transfer and store multiple copies of the same training data on the device. This is analogous to the redundancy problems faced with conventional hypervisors running many copies of the same operating system on a single server [162].

To avoid these pitfalls and provide efficient sharing, we need an approach that is aware of common I/O and computing processes among models that share a device. We consider a scheme, packing models, where multiple static neural network architectures (e.g., ones that are typically used in computer vision) can be rewritten as a single concatenated network that preserves the input, output, and backpropagation semantics through a `pack` primitive. Not only do such concatenations facilitate the partitioning of a single device they also allow us to synchronize data processing on GPUs and collapse common variables in the computation graph. It is often the case during hyperparameter tuning that the same model with various hyperparameter configurations are trained, and `pack` can feed a single I/O stream of training features to all variants of the model. In contrast, an isolated sharing way (e.g., training models isolatedly in sequence) may lead to duplicated work and wasted resources.

One of the surprising conclusions of this paper is that packing models together is not strictly beneficial. Counter-intuitively, certain packing policies can perform significantly worse than whole-device baselines–in other words, training a packed model can be slower than the sum of its parts (i.e., training these ”parts” one by one). This paper studies the range of possible improvements (and/or overheads) for using `pack`. Further, we deploy `pack` to hyperparameter tuning and demonstrate that it can greatly improve the performance of hyperparameter tuning in terms of the time needed to find the best or the most promising model.

Our experimental results suggest: (1) There is a range of performance impact, spanning from 40% faster execution to 10% slower execution on a single GPU for packing two models over unpacking them for a single training step, and the improvement is scalable when packing more models. (2) The benefits of the `pack` primitive largely depend on a number of factors including memory capacity, chip architecture, neural network structure, batch size, and data preprocessing overlap. (3) There exists a trade-off between

packing and unpacking when training multiple neural network models on limited resources. This trade-off is further complicated by architectural properties that might make a single training step bounded by computation (e.g., backpropagation is expensive) or data transferring (e.g., transferring training batches to GPU memory). (4) The `pack` primitive can speedup hyperparameter tuning by up to 2.7×.

2.1 Background

2.1.1 Motivation

Figure 2.1 demonstrates the typical data flow in neural network model training with stochastic gradient descent (or related optimization algorithms). We use the term *host* to describe CPU/Main-Memory/Disk hierarchy and *device* to refer to the GPU/Device Memory. In this setup, all of the training data resides on the host. Considering the typical training setup on the left side, a batch of data is taken from the host and copied to the device. Additionally, it is common in machine learning (especially in Computer Vision) that this data is preprocessed before it is transferred. Then, on the device, the execution framework calculates a gradient using backpropagation. Finally, using the results from the backpropagation, the model is updated.

In the typical "multiple-tasks-single-device" mode, resource sharing is often temporal–where one training task uses the whole GPU first and then switches full control to another task. Resource sharing in single mode is wasteful if the models are small and there is sufficient GPU memory to fit both models on the device simultaneously.

The right side of Figure 2.1 motivates a different solution. It allows multiple models to be placed on a single GPU. This packed mode could bring some potential benefits. Suppose we are training two models on the same dataset to test if a small tweak in neural network architecture will improve performance. The same data would have to be copied and transferred twice for training. If the system could pack together models when compatible in size, then these redundant data streams can be fused together to improve performance.

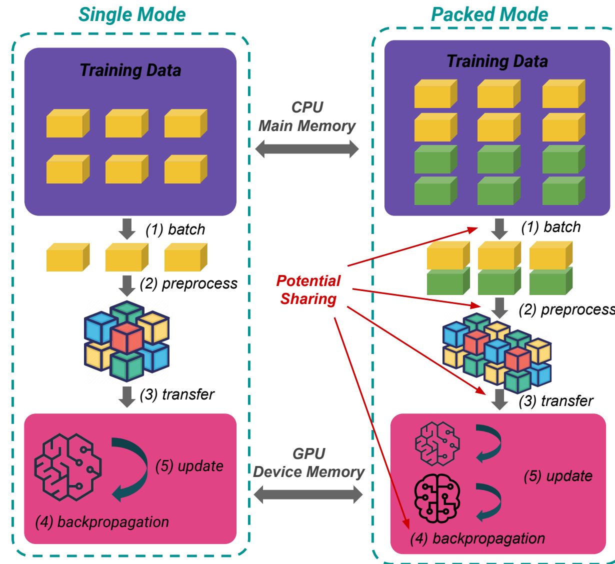


Figure 2.1: The dataflow of a training step in the single mode v.s. the packed mode. The training data resides in main memory and is copied over to the device in batches during each training step, resulting in a backpropagation computation and then a parameter update. By synchronizing the dataflow, the packed mode can reuse work when possible.

2.1.2 Basic Framework API

We desire a framework that can pack models together and jointly optimize their respective computation graphs when possible to reduce redundant work. We assume that we have access to a full neural network description, as well as the weights of the network. Each training task is characterized by four key traits: (a) *Model.* A computation graph architecture of the model with pointers to the input and output, equipping with some training hyperparameters (e.g., learning rate, optimizer, etc.) and assigning to a logical name that is uniquely identified. (b) *Device.* The target device to be used for placing and training models. (c) *Batch Size.* The batch size used in the training process, where each batch refers to the size of input data used in a single training step. (d) *Training Step.* The number of steps to train the model is also relevant to the number of epochs since one epoch typically consists of numerous steps.

Our objective is the following isolation guarantee: given these four traits, our framework will train the models in a fine-grained way but preserve the accuracy as if the training tasks were trained isolatedly and sequentially on a dedicated device. No action that the framework takes should affect training accuracy.

Such a framework requires three basic primitives: `load`, `free`, and `pack`. Users should be able to interact with our framework without worrying about exactly how the resources are allocated and on which devices the models are placed.

The primitives `load` and `free` can "copy in" and "copy out" models. Given a device name and model, `load` places the model on the device:

```
load(model, device)
```

Given a device name and model, `free` retrieves the model and frees the resources taken by the model:

```
checkpt = free(model, device)
```

State-of-the-art neural network training algorithms have additional states as a part of the optimizer. This state is stored with the model (see our experiments on computer vision models with optimizers in Section 2.3). Then, the API provides the primitive `pack` for packing. Suppose we have two neural network models:

```
output1 = nn1(input1)
output2 = nn2(input2)
```

The `pack` primitive combines both models into a new neural network by concatenating the output layers:

```
[output1 output2] = packed_nn([input1 input2])
```

This packing operation is fully differentiable and preserves the correctness semantics of the two original networks. Crucially, this allows the execution layer to process inputs simultaneously.

Thus, the models can be jointly trained using `pack`. The training steps have to be synchronized in the sense that the models are differentiated and updated at the same time. This synchronization leads to a complex performance trade-off, if the models are too different, the device may waste effort stalling on one model while either updating or differentiating on the other. This means that training a packed model may

be significantly slower than sequentially training each constituent model in it. However, the overheads from stalling may be counteracted by the benefits of reducing redundant computation. Navigating this complex trade-off space is the motivation for this study, and we seek to understand under what conditions `pack` beneficial.

2.2 Implementation

We build a prototype system on top of TensorFlow to implement the above framework APIs and take image classification as our motivating application in our implementation, but the idea of `pack` is generally compatible with other platforms and applications.

2.2.1 Packing

`Pack` is a lossless operation that concatenates the outputs of two or more neural network models. Since it is lossless, it preserves the forward and backward pass semantics of the model. The basic operation can be written as packing multiple output variables, as illustrated in the following example:

```
mlp_out = #reference to mlp output
resnet_out = #reference to resnet output
densenet_out = #reference to densenet output
packed_out = pack([mlp_out resnet_out densenet_out])
```

This `packed_out` can be thought of as a new neural network model that takes in all input streams (even possibly different input data types) and outputs a joint prediction. Thus, we can do everything to a packed model that we could do to a single neural network. The packed model can be differentiated and the model parameters can be updated iteratively. The model can be placed on a device, such as a GPU or TPU, as a single unit.

While this gives us scheduling flexibility, there is a major caveat. By packing the models together, we create an artificial synchronization barrier. If one of the models is significantly more complex than the

others, it will block progress. Likewise, if one of the operations saturates the available compute cores, progress will stall as well. Naive packing leads to a further issue where the input batch has to be synchronized in dimension as well (each model is differentiated or evaluated the same number of times). Therefore, without further thought, the scope of `pack` is very narrow.

2.2.2 Misaligned Batch Sizes

Requiring that all packed models have the same batch size is highly restrictive, but we can relax this requirement. Our method is to rewrite the packed model to include a dummy operation that pads models with the smaller batch size to match the larger ones in dimension. The `pad` primitive is exploited for packing models with different batch sizes. The original models are packed and trained based on the batch with the largest size, but the batches for the models with smaller batch sizes will be padded. During training and inference, the padding is sliced.

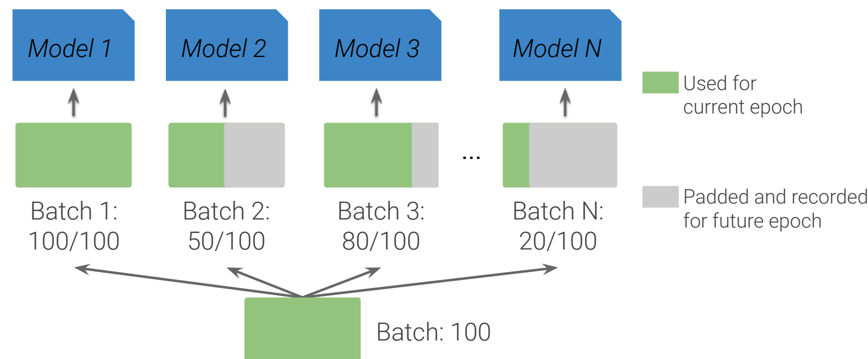


Figure 2.2: All the models share the batch input stream, each batch is padded and sliced for training the packed model.

As depicted in Figure 2.2, there is a set of original models with various batch sizes, and the largest training batch size (i.e., 100) is selected and fed to the packed model for a single training step accordingly. Then, the batch will be replicated for n original models in the packed model. The model 1 takes the entire batch, whereas the replicated batches $2, \cdots, N$ are sliced to match the models' requirement. Thus, all the models can be trained together.

Simply slicing may result in statistical inefficiency since only a fraction of the entire dataset is used

during each epoch for the models with smaller batch sizes. To address this issue, we track the progress of each model individually to ensure that there is no loss in training datasets. Assuming we train the packed model in Figure 2.2 for one epoch. When model 1 finishes training and is unloaded, model 2 achieves 50% progress and uses 50% of the training dataset, model 3 has been training using 80% dataset, and so do the other models (dataset usage is recorded for all models). Then, the packed model takes the current largest batch size (i.e., 80 from model 3) and uses the rest training dataset of model 3 to train the packed model. Due to slicing, it is obvious that the unused training datasets of model 3 are included in the unused datasets of other models. The process continues until all models are trained completely and thus no training data is missing.

2.2.3 Misaligned Step Counts

Another issue with synchronization is that different models may need to be trained for a different number of steps. Even if all of the models are the same, this can happen if the user is trying out different batch sizes.

We use `load` and `free` to address this issue. As demonstrated in Figure 2.3, we train three models with batch size 20, 50, and 100 for one epoch using $10,000$ images and labels, and they require 500 steps, 200 steps, and 100 steps. We pack them for training, and when the model with 100 steps is finished, it is freed and checkpointed. Then, a new model can be loaded and packed with the incomplete models to continue training. This mechanism may bring an overhead of loading models but can support training models with a different number of steps.

2.2.4 Eliminating Redundancy

`Pack` forces synchronization, which means that dimensional differences between the models or training differences between the models can lead to wasted work. However, `Pack` can allow the system to eliminate redundant computations and data transfers. Consider a hyperparameter tuning use case where we are training the same network with a small configuration tweak on the same dataset:

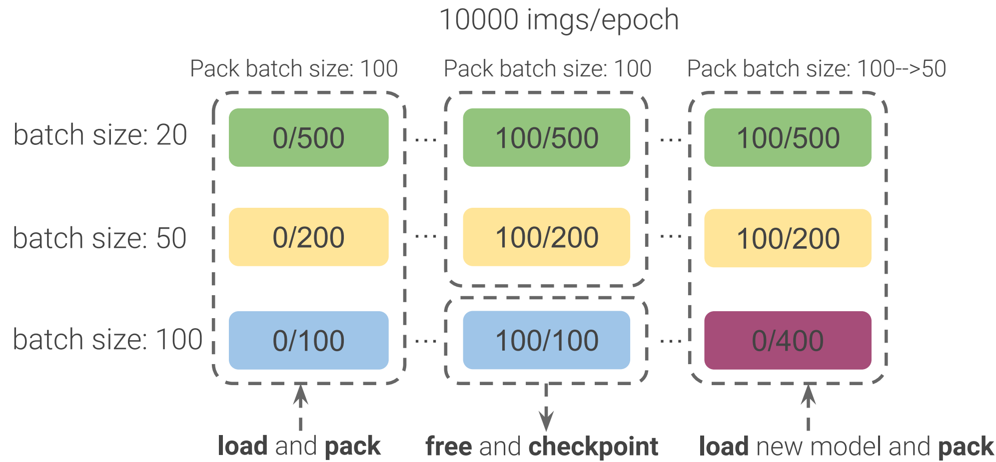


Figure 2.3: Early finished model is freed and checkpointed, and a new model is packed with the others for further training.

```
nn_conf1_out = nn_conf1(input1)

nn_conf2_out = nn_conf2(input2)
```

In this case, `input1` and `input2` refer to the same dataset. We can avoid transferring the batch multiple times by symbolically rewriting the network description to refer to the same input:

```
nn_packed_out = pack([nn_conf1_out nn_conf2_out])

[output1 output2] = nn_packed_out(input)
```

The potential upside is significant as it reduces the amount of data transferred along a slower I/O bus. Furthermore, eliminating redundant computation goes beyond identifying common inputs. Preprocessing is a common practice for machine learning training tasks. The preprocessing operations (e.g., data augmentation, image decoding) happen before training and can actually dominate the total execution time of some models. When packing models that take the same preprocessing, the `pack` primitive can fuse the steam processing and eliminate redundant tasks. This idea can be extended if multiple models have fixed featurization techniques or leverage the same pre-trained building blocks.

2.3 Profiling Model Packing

As it stands, model packing leads to the following trade-offs. Potential performance improvements include: (1) eliminating redundant data transfers when models are trained on the same dataset, (2) combining redundant computations including preprocessing, and (3) performing computations (forward and back propagation) in parallel if and when possible. On the other hand, the potential overheads include (a) models that dominate the device resources and block the progress of the others, (b) overhead due to misaligned batch sizes, and (c) overhead due to loading and unloading models with a differing number of training steps.

This section describes a series of experiments that illustrate when (what architectures and settings) packing is most beneficial.

2.3.1 Profiling Setup

Our server is 48-cores Intel Xeon Silver 4116@2.10GHz with 192GB RAM, running Ubuntu 18.04. The GPU is NVIDIA Quadro P5000. Our evaluation uses four models: Multilayer Perceptron with three hidden layers (MLP-3), MobileNet [135], ResNet-50 [55], and DenseNet-121 [61] – with all models implemented in TensorFlow 1.15. The default training dataset is $10,000$ images from ImageNet [133], and the required input image size of each batch is 224×224, which is commonly used. Batch sizes start from 32 and go up to 100 in the experiments [15].

In our experimental methodology, the first training step is always omitted for measurement due to the CUDA warm-up issue, and the measurement of the single step excluded loading time. We only measure the loading cost to investigate whether it dominates the performance (middle column in Figure 2.6). So, this measurement is orthogonal to any pipelining that might happen at a different level of abstraction. The results in the paper are averaged over 5 independent runs.

2.3.2 Profiling Metrics

We evaluate the `pack` primitive against three performance metrics defined as follows.

Improvement: We measure the time of a single training step of the packed model. Since one training epoch can be treated as a series of repeating training steps and a complete training process is made with multiple epochs, the single training step measurement can be used to estimate the overall training time. We denote the step time as T_s and assume that there are n models (model $1, \cdots, n$), and we compare the time of a single training step in packed and sequential mode. We first train models $1, \cdots, n$ isolatedly and sequentially and measure the time of a single training step:

$$T_s(Seq) = T_s(\text{Model } 1) + \cdots + T_s(\text{Model } n) \tag{2.1}$$

Then, we pack these models for training and measure the single training step, which is defined as follows:

$$T_s(Pack) = T_s(Pack(\text{Model } 1, \cdots, n)) \tag{2.2}$$

Thus, we define the improvement metric as follows:

$$IMPV = \frac{T_s(Seq) - T_s(Pack)}{T_s(Seq)} \tag{2.3}$$

The improvement metric can quantify the benefits brought by `pack` primitive, and comparing $IMPV$ of various training setups can identify performance bottlenecks.

Memory: Fine-grained resource sharing (e.g., training multiple models together on a single device) requires sufficient device memory; thus, measuring the memory usage of the packed model can provide insights for scheduling different models given a specific device memory capacity. We evaluate the peak of memory usage over the training epoch. This is because if the usage peak exceeds the GPU memory capacity, the training process will be terminated due to a GPU memory error. We measure the allocated memory and not the active memory used.

Switching Overhead: Training the models isolatedly and sequentially on a single device can bring an additional switching overhead. For example, the GPU has to unload the old model and the associated

context and then load the new models and prepare the context. `Pack` significantly reduces such overhead since packing models suffer from model switching less often (multiple models can be trained together given enough GPU memory so that loading and unloading operations can be avoided). The switching overhead is measured through the following method: We train n models isolatedly and sequentially for one epoch and capture the training time, which is denoted as $T_e(Seq)$. Then, we train n models individually and denote $T_e(Model)$ as the training time of one epoch for each model. Thus, the switching overhead of training n models is defined as:

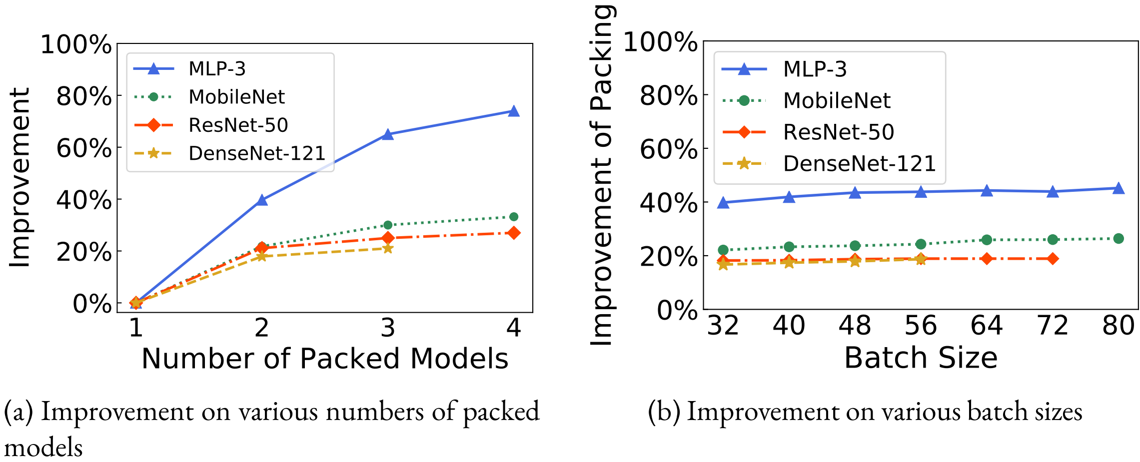
$$SwOH(n) = T_e(Seq) - T_e(\text{Model 1}) \cdots - T_e(\text{Model n}) \tag{2.4}$$

However, we hypothesize that the overhead amortizes over an entire training procedure. This is because $SwOH$ depends on the number of models instead of the number of training steps and epochs. Since training a model usually involves numerous training steps and many training epochs, compared with much longer training time, the switching overhead is minor (section 2.3.5).

2.3.3 Improvement

We evaluate packing performance as a function of batch size and the number of models. Figure 2.4a shows that as the number of packed models increases, so do the relative benefits until the resources are saturated. The line of DenseNet-121 ends early because packing four DenseNet-121 takes too much GPU memory and results in an Out-Of-Memory (OOM) issue. However, the potential for resource savings is significant. If one is training multiple MLP models, there can be up to an 80% reduction in training time. In short, *it is wasteful to allocate entire devices to small models.*

Figure 2.4b illustrates the relationship between batch size and relative improvement when packing two models. The lines of ResNet-50 and DenseNet-121 both end early because the OOM issue emerges when the batch size goes to 80 and 64, respectively. These models are mostly GPU-compute bound. Increasing the batch size has a negligible improvement in time, even if the packing setup can combine the data transfer. We will see that this story gets more complicated when considering preprocessing.



(a) Improvement on various numbers of packed models

(b) Improvement on various batch sizes

Figure 2.4: *Improvement* of packing models when increasing the number of models and batch size on GPU. The Y-axis indicates the reduction in training time compared to sequential execution (as defined in Equation 2.3).

2.3.4 Memory Usage

We track the GPU memory usage of training individual models and packed models with different batch sizes for one epoch. We particularly care about the memory peak and whether it is beyond the memory capacity.

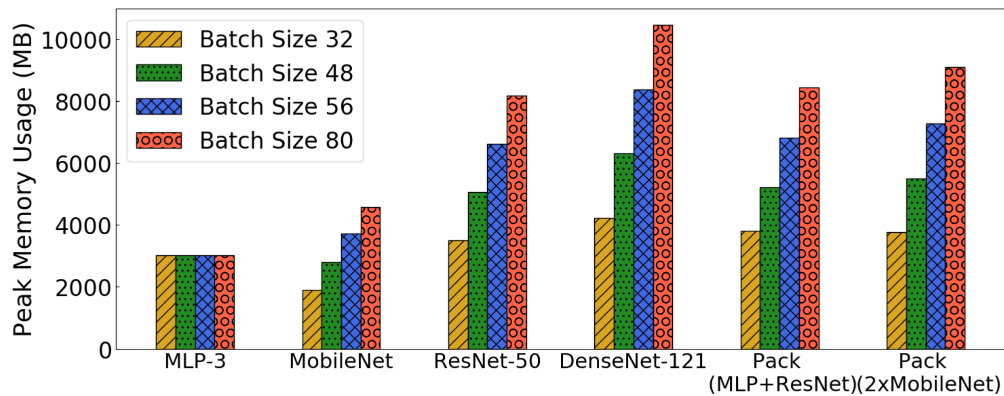


Figure 2.5: GPU memory peak of different models

As depicted in Figure 2.5, for convolutional neural networks like ResNet, MobileNet, and DenseNet, the GPU memory usage is proportional to the batch size as more intermediate results will be stored as batch size increases. Similarly, when packing two models, the GPU memory usage is the sum of memory usage. However, the GPU memory usage peak of MLP-3 model remains the same as the

batch size goes up. This is mainly due to two reasons: (1) we find that for simple models, TensorFlow's greedy memory allocation policy over-allocates more GPU's memory when the actual usage is lower than a specific threshold; (2) the majority of computations for MLP-3 are dot products and are placed on CPU by TensorFlow and do not occupy much GPU memory. More specifically, without any annotations, TF automatically decides whether to use the GPU or CPU for an operation [152] (we also used the TF profiler to trace the training process and found the majority of operations in MLP-3 are placed on the CPU). Thus, GPU memory usage of a single MLP-3 remains the same due to the pre-allocation.

2.3.5 *Switching Overheads*

We profile the switching overhead of two models to illustrate how much the overhead can accumulate as more models are trained (the time `pack` can save).

	Model	$T_e(Seq)$	$T_e(Model)$	$SwOH(2)$
GPU	MLP-3	133s	61s	11s
	MobileNet	227s	107s	13s
	ResNet-50	274s	130s	14s
	DenseNet-121	305s	144s	17s

Table 2.1: $SwOH$ of training two models sequentially

As shown in the Table 2.1, albeit the accumulation, the switching overhead (using Equation 2.4) is minor compared to the overall training time, and it is even negligible when more epochs are involved in a training process. This also confirms our hypothesis of the switching overhead.

2.3.6 *`Pack` vs CUDA Parallelism*

Current NVIDIA GPUs support executing multiple CUDA kernels in parallel at the application level. Thus, we conduct an experiment under the same environment as we used in the paper to train models in parallel at the CUDA GPU kernel. We run multiple simultaneous training processes on TensorFlow. We evaluate this method in the experiments where two processes are boosted at the same time to train

the same models (MLP, MobileNet, ResNet, DenseNet) with the same optimizer and same batch size (ranging from 32 to 100).

Although CUDA supports it, our results show that it is not an efficient technique. When the models train on the same data, parallel training in isolated kernels leads to duplicated I/O and duplicated data in memory. In the image processing tasks that we consider, the training data batch takes up a substantial amount of memory. We find that in all but the simplest cases leads to an OOM error: "failed to allocate XXX from device: `CUDA_ERROR_OUT_OF_MEMORY`". We find similar results when the models train on *different* data—as there is duplicated TensorFlow context information in each of the execution kernels. This error happens in all the above experiments except packing the MLP model (due to its lightweight size).

Even with the MLP model, the `pack` primitive shows benefits at scale. For instance, the training time of a single step based on CUDA parallelism is 184ms for both two processes, and the packing method takes 200ms. However, as the batch size is increased to 100, the former one takes 1660ms, while the latter one costs 1500ms. We interpret these numbers as an indication that the `pack` primitive incurs smaller context overhead over the native CUDA parallelism at the application level.

2.3.7 Ablation Study

To further evaluate the performance of packing models on GPU, we test more cases based on the five factors: (1) whether the models have the same architecture; (2) whether the models share the same training data; (3) whether the models take the preprocessed data or raw data for training, i.e., if the preprocessing is included in training; (4) whether the models use the same optimizer; and (5) whether the models have the same training batch size. In this ablation study, we follow the configurations illustrated in Table 2.2 and evaluate the `pack` primitive. Without loss of generality, we focus on packing two models to understand the relationship between the training time and the above factors, packing more models follows the trends as demonstrated in Figure 2.4.

Figure 2.6 presents the results of the ablation study. In the figure, the data points in each sub-

Factor	Config	Description
Model	Same	Packing two same models
	Different	Packing two different models. We evaluated MLP-3 vs. MobileNet, MobileNet vs. DenseNet-121, ResNet-50 vs. MobileNet, and DenseNet-121 vs. ResNet-50.
Training Data	Same	All packing models take the same training batch data
	Different	All packing models take the different training batch data.
Preprocess	Yes	Preprocessing is included in each training step. Training batches are raw images (e.g., JPEG) transferred from disk to GPU.
	No	Preprocessing is excluded in each training step. Training batches are preprocessed and formatted before transferring to GPU.
Optimizer	Same	All packing models use the same optimizer.
	Different	All packing models use different optimizers.
Batch Size	Same	All packing models take the same batch size.
	Different	The packing models may take the different batch sizes for a single training step, e.g., one is 32 batch size, the other is 50 batch size.

Table 2.2: Model configurations for ablation study

figure represent the $T_s(Seq)$ and $T_s(Pack)$ of various configurations with fixed one configuration (e.g., same batch size or same model). The red point (triangle pointed down) indicates that packing two models brings more overhead compared with training them sequentially with this configuration, i.e., $T_s(Pack) > T_s(Seq)$, while the green point (triangle pointed up) means the opposite. The further from the line, the more significant the performance difference.

As we can see from Figure 2.6, the best scenarios are where the same training data and the same batch size are used. Through all the configurations, the `pack` primitive always brings benefits when we train models with the same data since it will reduce the data transfer. Similar benefits happen with the same batch size configuration. This is important to note because even when the same models are trained but with different data inputs and batch sizes, there can be significant downsides to packing. It is not simply a matter of looking at the neural network architecture, but the actual training procedure factors into the decision of packing.

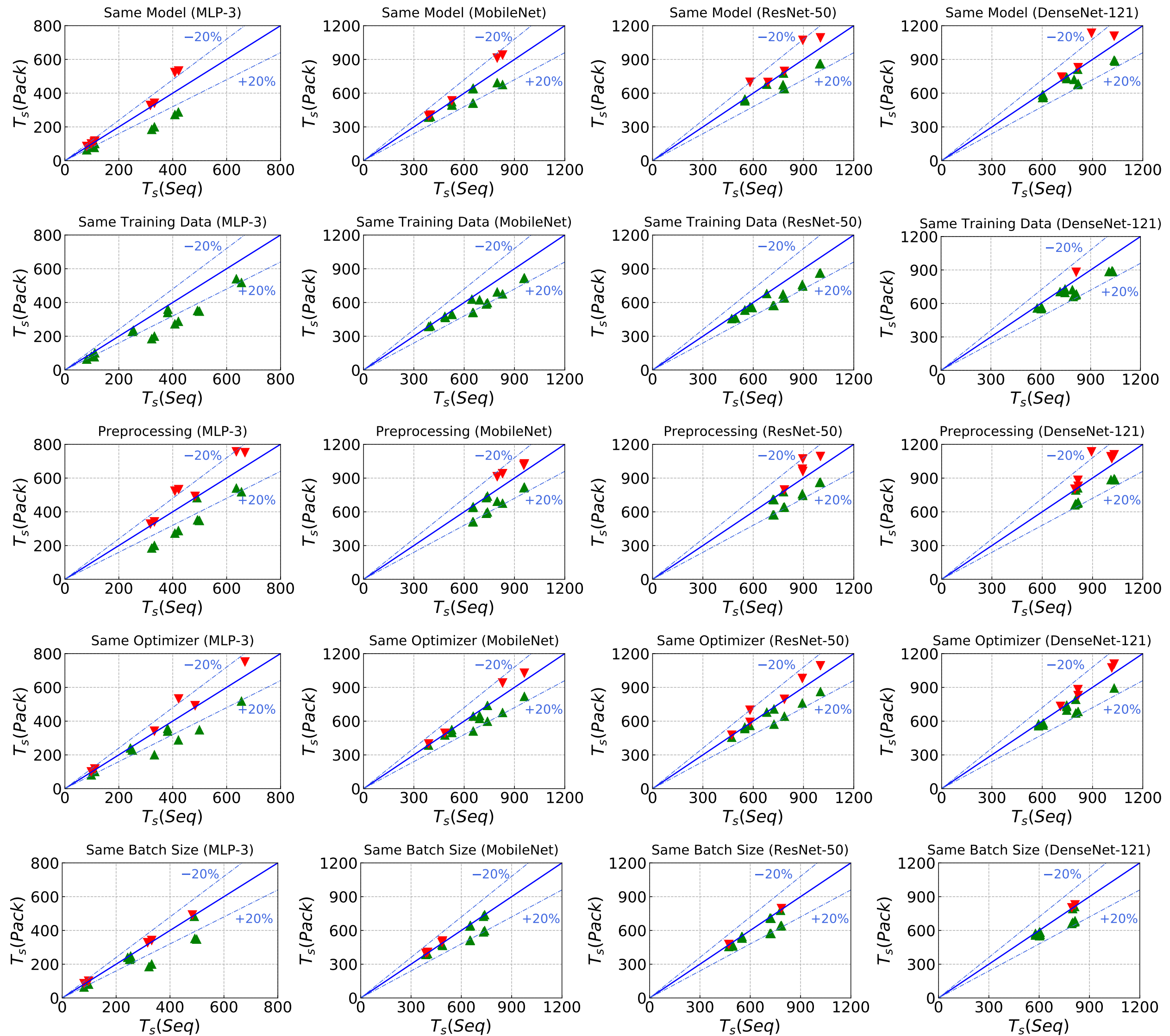


Figure 2.6: $T_s(Seq)$ vs. $T_s(Pack)$ (milliseconds) when packing two models on a GPU

2.4 Pack-Aware Hyperparameter Tuning

We demonstrate how pack benefits hyperparameter tuning in this section. As we show in the previous section, pack brings the biggest improvement when the models trained are similar and train on the same input data. Such a scenario naturally arises in hyperparameter tuning. Developers have to search over adjustable parameters such as batch sizes, learning rates, optimizers, etc. Tuning such hyperparameters is crucial to finding models that generalize to unseen data and achieve promising accuracy.

There are a number of real-world scenarios where multiple models are trained on the same data, and

we demonstrate hyperparameter tuning as a representative application. Furthermore, `pack` is a *simple but practical* mechanism that can be implemented at the application level, which allows for a wide variety of deployment scenarios.

2.4.1 Hyperband

We explore extending a state-of-the-art hyperparameter tuning algorithm, Hyperband [89], to better share GPU resources. Hyperband works by repeatedly sampling random parameter configurations, partially training models with those configurations and discarding those configurations that do not seem promising. Prior work suggests that Hyperband is effective for parallel hyperparameter search in comparison to sequential algorithms such as Bayesian Optimization [87].

Hyperband poses the search as an online resource allocation problem. Given N discrete model configurations to test, it partially trains each configuration and discards those that do not seem promising based on a technique called successive halving. The search routine follows the structure of Algorithm 1.

Algorithm 1: Hyperband

input : R, η
output: Conf with the smallest intermediate loss so far
for $r \leftarrow 0$ **to** $\lfloor log_\eta(R) \rfloor$ **do**
  Randomly sample T from N confs without replacement;
  for $i \leftarrow 0$ **to** r **do**
    Train *conf* i for multiple epochs;
    Calculate the intermediate loss of *confs* i;
    Keep a fraction of the best *confs* for the next iteration;

Intuitively, Hyperband only allocates resources to the most promising configurations. At the maximum iteration, the most promising configurations are trained for the longest. This basic loop can be trivially distributed as a random partition of N configurations. Although Hyperband is able to optimize the process of hyperparameter tuning, the algorithm is long-running since it consists of a large number of trial hyperparameter configurations to run, and each of them usually occupies the entire GPU resource when running.

2.4.2 `pack`-aware Hyperband

Our `pack` primitive allows Hyperband to jointly train configurations when possible, thereby reducing the overall training time. We propose a `pack`-aware Hyperband that leverages model packing to improve its performance when there are more models to evaluate than available GPU devices. The challenge is to determine which configurations to train jointly and which to train sequentially.

For each iteration, our `pack`-aware Hyperband will partition sampled T models to multiple packed groups that can fit on a single device (the size of a packed group does not exceed the amount of memory of the GPU). Then, the optimization problem is to search over all packable groups to find the best possible configuration (one that maximizes the overall run time). Note that singleton partitioning (every single model forms a group) is always a viable solution and potentially even an optimal solution in some cases. We call this primitive `pack_opt`, which solves the search problem by producing feasible packing groups and identifying the most promising configuration. Accordingly, we can run a modified Hyperband loop that packs models when beneficial, as shown in Algorithm 2.

There are two challenges in `pack_opt`: *(C1)* developing an accurate cost model to evaluate the cost of a packed plan, and *(C2)* a search algorithm that can effectively scale with T. Of course, the combinatorial nature of this problem makes both *(C1)* and *(C2)* hard to accomplish optimally and we need a heuristic to address this problem. Recognizing that similar models could be packed well together, we design a nearest-neighbor-based heuristic.

The method randomly selects a single configuration (out of T for each round) as the centroid and packs all other configurations similar to it until the device runs out of memory. This process is repeated until all models are packed or determined that the best choice is to run them sequentially. For calculating the similarity, we map hyperparameter configurations to multi-dimensional feature space and measure the pairwise Euclid distance among all the configurations. A user-tuned similarity threshold decides how aggressively the system will pack models. For example, considering the sampled hyperparameter configurations shown in the Table 2.3, we take the standard distance unit as 1, and compute the distance between any two configurations. For categorical hyperparameters like optimizer and activation, the distance is 0

if same and 1 if different, for numeric hyperparameters, we use the index to compute distance. So, the distance between configuration A [batch size:20, optimizer: SGD, learning rate:0.01, activation: ReLu] and configuration B [batch size:40, optimizer: Adagrad, learning rate:0.01, activation: ReLu] is 5.

Algorithm 2: `Pack`-aware Hyperband

```
input  : R, η
output: Conf with the smallest intermediate loss so far
for s ← 0 to ⌊log_η(R)⌋ do
    Randomly sample T from N confs without replacement;
    packed_group ← pack_opt(T);
    for g ← 0 to packed_group do
        Train packed_conf g for multiple epochs;
        Calculate the intermediate loss of packed_conf g;
        Keep a fraction of the best confs for the next iteration;
```

Despite being imperfect, Euclid distance has been proven to be a practical metric. We also applied a pairwise Training Time-based distance that reflects the importance of all features using the training time metric. Specifically, we train two configurations in a packed way and a sequential way for a single step, respectively, measuring the training time and calculating the difference with normalization. We take the difference as the distance and deploy it to the pack-ware hyperband. Our empirical experiments show that the Euclid distance method is still faster than the training time-based distance method up to 18%.

Note that since the main benefit of `pack` comes from sharing and padding the input, packing different models can still improve the performance. So, `pack` is performant in the exploration phase of various hyperparameter tuning methods. Taking Bayesian Optimization as an example, in its exploration phase, the hyperparameter configurations are sampled and evaluated for some predefined objective functions. Thus, the sampled configurations can be packed during the exploration for acceleration.

2.4.3 Evaluation for Hyperparameter Tuning

The goals of our evaluation are two-fold: first to demonstrate that `pack` can significantly improve hyperparameter tuning performance and second to evaluate our `pack`-aware Hyperband. We conduct the experiments based on the same hardware environment as illustrated in section 2.3.1. We examine the Hy-

perband variants on CIFAR-10 [76] which consists of 60000 color 32×32 images in 10 classes (50000 for training dataset, 10000 for testing dataset). The system's goal is to find the best configuration of those described in Table 2.3, thus all hyperparameter configurations are from the combination of all hyperparameters which has 1056 configurations in total. The input, R and η, are set to 81 and 3, according to the original Hyperband paper [89].

Hyperparameter	**Value**
Batch size	20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70
Optimizer	Adam, SGD, Adagrad, Momentum
Learning Rate	0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1
Activation	Sigmoid, Leaky ReLu, Tanh, ReLu

Table 2.3: Hyperparameter configurations for evaluation

We also compare our pack-aware Hyperband against two other heuristics:

Random Pack Hyperband: After sampling hyperparameter configurations, the method randomly selects m configurations to pack and evaluates them together, then it keeps the best n configurations and discards the rest as the original Hyperband does.

Batch-size Pack Hyperband: Rather than randomly selecting, *Batch-size Pack Hyperband* only packs the models with the same batch size. Although the number of packed models is confined by GPU memory size, a greedy method is employed (i.e., packing as many models as possible until full usage of GPU memory).

We evaluate the overall running time of Hyperband with the different pack_opt algorithms. As presented in the Table 2.4, all the pack-aware Hyperband variants can reduce the running time w.r.t the original Hyperband algorithm for all scenarios. Our proposal, *kNN Pack Hyperband*, achieves the best performance since it takes advantage of our findings from the previous section where packing the most similar models leads to the biggest improvements. The conclusion is that such an approach can save time (and consequently money) in real end-to-end tasks. A simpler heuristic, *Batch-size Pack Hyperband*, is not as effective because it under-utilizes the available GPU resources by missing packing opportunities with

models with slightly different batch sizes. To emphasize this point, a *Random Pack Hyperband* can save more time than *Batch-size Pack Hyperband* since it achieves better GPU resource utilization. Our kNN strategy gets the best of both worlds: it finds the most beneficial packing opportunities while completely utilizing the available resources, and benefits are scalable when deployed in an environment with a larger GPU resource.

	Original	**Batch-size**	**Random**	**kNN**	**Speedup**
MLP-3	9236s	5260s	3682s	3491s	$\sim$ **2.7**$\times$
MobileNet	52092s	45787s	36973s	30182s	$\sim$ **1.7**$\times$
ResNet-50	98067s	89162s	75436s	70047s	$\sim$ **1.4**$\times$
DenseNet-121	131494s	126437s	117405s	108673s	$\sim$ **1.2**$\times$

Table 2.4: Performance of `pack`-aware Hyperband

2.5 Related Work

There are a number of systems that attempt to control resource usage in machine learning, specifically memory optimization [121, 164, 176, 80, 66, 138, 172, 18], but we see this problem as complementary. For example, `pack` is similar in mechanism to a recent proposal, HiveMind [121], where multiple models are fused into a single computational graph during training. However, we additionally contribute: (1) a cost model and optimizer that decides when this fusion is most beneficial, (2) integration with a hyper-parameter tuning algorithm to demonstrate end-to-end improvements over a training workload, and (3) a data batching scheme that allows packing models with different batch sizes without hurting statistical efficiency. These contributions are noted as existing limitations in HiveMind. We also discuss the related works that study hyperparameter tuning systems and multi-tenancy systems in machine learning.

2.5.1 Systems for Hyperparameter Tuning

Since hyperparameter tuning is a crucial part of the machine learning development process, a number of systems have been proposed to scale up such search routines. For example, Google Vizier [45] exposes

hyperparameter searching as a service to its organization's data scientists. Aggressive "scale-out" has been the main design principle of Vizier and similar systems [92, 87, 89].

Recently, there has been a trend toward more controlled resource usage during hyperparameter tuning. Cerebro borrows the idea of multi-query optimization in database systems to raise resource efficiency [117]. HyperSched proposes a scheduling framework for hyperparameter tuning tasks when there are contended specialized resources [91]. And, some work has been done on resource management [143] and pipeline re-use [88] in the non-deep learning setting. We believe that `pack` and `pack_opt` are two primitives that are useful in hyperparameter tuning when specialized hardware such as GPUs and TPUs are limited in usage. Also, although our experiments focus on hyperparameter tuning, `pack` and `pack_opt` primitives can be easily extended to other scenarios.

2.5.2 Systems for Multi-tenancy

Most current projects about building multi-tenant systems for machine learning deployment are based on device-level placement, i.e., dividing resources at the granularity of full devices (e.g., an entire server or GPU). Here, the scheduler partitions a cluster of servers where each server has one or more GPUs for various model training tasks and seeks to reduce the overall training time by intelligent placement. Other scheduling methods have followed, such as Tiresias [48] and Optimus [125]. Several extensions have been proposed to this basic line of work, including fairness [107], preemption [171], and performance prediction [180]. Gandiva is a cluster scheduling framework for deep learning jobs that provides primitives such as time-slicing and migration to schedule different jobs. CROSSBOW is a system that enables users to select a small batch and scale to multiple GPUs for training deep learning models [74]. PipeDream is a deep neural network training system for GPUs that parallelizes computation by pipelining execution across multiple machines that partitions and pipelines training jobs across worker machines [120]. Ease.ml is a declarative machine learning service platform that focuses on a cost-aware model selection problem in a multi-tenant system. [90]. Some recent works also exploit data parallelism to accelerate the training process. MotherNets can ensemble different models and accelerate the training process by

reducing the number of epochs needed to train an ensemble [165]. FLEET theoretically proves that optimal resource allocation in deep learning training is NP-hard and proposes a greedy algorithm to allocate resources [49].

Compared with these previous works, our prototype implements a method to pack diverse models with different batch sizes. We also conduct a comprehensive evaluation that differentiates performance wins from variable elimination v.s. improved utilization, and highlights the potential for packed models to train slower than the sum of their parts, which is only apparent with modern architectures. Taking these inspirations, we further deploy our primitives to hyperparameter tuning and show performance improvement.

2.6 Discussion

Our core contribution is demonstrating the potential benefits (and overheads) of combining similar models into a single computational graph, and thus collapsing common data inputs during training iterations. This was the reason why we chose not to optimize this process at a lower level (e.g., MPS/Hyper-Q [19]), where we found that the majority of benefits could be attributed to simply sharing common inputs and context variables. Thus, the key goal of our proposed optimizer is to decide whether two models share enough to see a potential benefit and control the exact execution order of the computation is orthogonal to our contribution.

Our long-term goal is to build a system for multi-tenant deep learning deployment, and we believe the `pack` will be one of the core parts of multi-tenancy systems for machine learning. In hyperparameter tuning there is a single user and a clear SLO (find the best model configuration overall), then to extend to more general multi-tenancy settings where concurrent models are trained, we will reason about multiple users, priorities, and user-specified objectives. For this, we decide to first make a deep investigation on a single GPU so that we will know how to optimize when there are multiple GPUs. Thus, any distributed training and regarding optimization is out of the scope of the paper.

We implemented `pack` on TensorFlow to conduct a comprehensive evaluation and highlight its ben-

efits in hyperparameter tuning. Although we believe that a custom execution platform could improve performance, `pack` doesn't require the modification of any specific framework and can be implemented across frameworks. We focus `pack` as a higher-level primitive due to (1) the optimizations will be more transferable across ML execution frameworks and thus increase the impact or applicability of our insights, and (2) many low-level libraries are highly optimized and introduce these changes (e.g., supporting jagged arrays) we believe are interesting research questions on their own.

CHAPTER 3

ROTARY: A RESOURCE ARBITRATION FRAMEWORK FOR PROGRESSIVE ITERATIVE ANALYTICS

A growing concern for organizations is high resource usage in data analytics [75, 146, 33, 141]. 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, 53, 4, 43, 110]. 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.

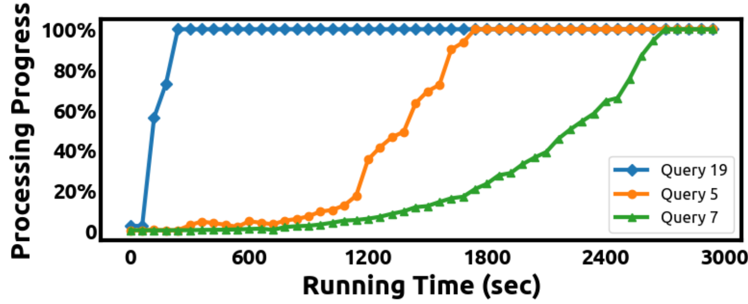
A traditional job scheduler places an immense level of trust in a user's ability to decide when to terminate these progressive iterative analytic jobs appropriately, which may bring disastrous consequences. For example, consider a user training a convolutional neural network for a fixed time of 500 epochs. 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 de-

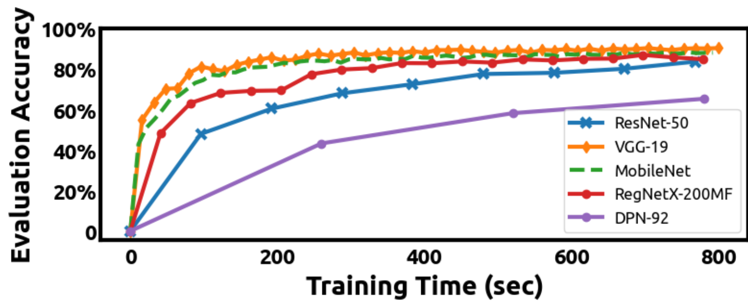
cisions need to consider a job's prioritization, specified completion criteria, the available resources, and other jobs waiting for the resources — a problem we call *resource arbitration*, which is a novel adaptive and completion criteria-aware scheduling paradigm. We identify two widely used applications 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 Figure 3.1 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 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 be completed 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.*



(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.

Figure 3.1: Progress curves of AQP and DLT jobs

We believe that resource arbitration and traditional scheduling systems [159, 34, 156, 57, 125, 168, 48, 65, 179, 161, 129, 101, 98] 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 Figure 3.2, 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 [89] 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.

Adaptive \ Completion Criterion	No	Yes
Yes	**Resource-oriented Scheduling:** Rayon[11], TetriSched[12], Trident[13], Optimus[14], HiveD[18], HaLoop[26]	**Resource Arbitration:** Rotary
No	**Time-Sharing Scheduling:** Round-robin **Dynamic Priority Scheduling:** Earliest-Deadline-First Least-Accuracy-First	**Progress-aware Scheduling:** ReLAQS [32] (accuracy-oriented)

Figure 3.2: 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 [10] 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 [151] 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.

3.1 Related Work

To the best of our knowledge, Rotary is the first resource arbitration system for DLT jobs to support user-defined completion criteria. Thus, we broadly review the related works to position our work.

3.1.1 Scheduling for AQP

Approximate query processing scheduling works are related to our framework since they focus on orchestrating the AQP jobs. However, to the best of our knowledge, there is not much work in this area [86, 26].

iOLAP is one of the representative works [174], which 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 and 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 is similar work to iOLAP lies in this area [2]. However, they mainly focus on scheduling query plans.

For scheduling AQP jobs, ReLAQS [144], which serves as one of the baselines in our experiments, is the state-of-the-art work. It is 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) ReLAQS only schedules CPU cores, Rotary-AQP further considers memory consumption when preempting resources; (2) 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; (3) Compared with ReLAQS, Rotary-AQP can support adaptive running cycling for short-running and long-running AQP jobs.

3.1.2 Scheduling for Machine Learning

We consider scheduling systems for machine learning as the most relevant works. We first review the works that define fixed scheduling objectives for DLT jobs. MArk 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 [175]. Some works like Tiresias [48] and Optimus [125] schedule machine learning jobs with time constraints.

Scheduling systems for machine learning are widely deployed as well. 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 [168]. 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 [65]. HiveD [179] is designed to be a Kubernetes scheduler extension for Multi-Tenant GPU clusters, which can guarantee resource reservation for DLT jobs. PipeDream [120] is a deep learning training system that schedules computation by pipelining execution across multiple machines to accelerate the training process. AntMan [169] 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 [129] is a resource-adaptive deep learning (DL) training and scheduling framework that 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 resource allocation and job preemption.

3.1.3 Multi-tenant Systems

Multi-tenant systems, which don't focus on job scheduling but also have been deployed for AQP and DLT applications, should also be mentioned.

BlinkDB [3] is an AQP system that is based on Apache Hadoop and devises effective strategies to select proper samples (offline generated) in distributed clusters to answer newly coming queries. Quickr [68] is designed for executing ad-hoc queries on big-data clusters that do not need any pre-computing of the whole dataset spread over the clusters. SnappyData [116] is a platform to support OLTP, OLAP, and stream analytics based on Apache Spark.

Multi-tenant systems for deep learning are also proposed and deployed recently. FfDL is a deep learning platform in IBM to support the multi-tenant distributed training of models based on Kubernetes [64]. Facebook also reveals some design choices for building a datacenter to handle multi-tenant training and inference, like the importance of co-locating data with computation [53]. Ease.ml is a declarative machine learning service platform that focuses on a cost-aware model selection problem in a multi-tenant system [110]. CROSSBOW [74] is a system that supports users to select a small batch and scale to multiple GPUs for deep learning training.

3.2 Resource Arbitration Framework

3.2.1 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 [86]. 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 [54] or Bi-LSTM [47]), 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.

3.2.2 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 Figure 3.3, 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.

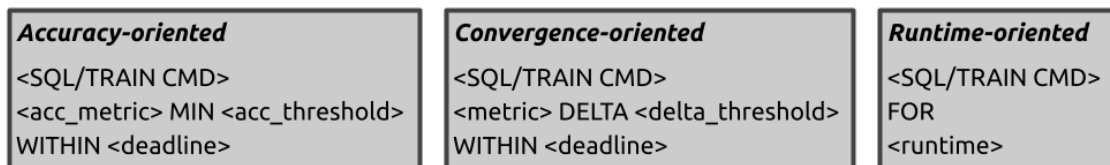


Figure 3.3: Templates of user-defined completion criteria

Figure 3.4 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, and 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.

Figure 3.4: 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 Figure 3.4, *ACC* is used for measuring convergence, but other metrics, such as *LOSS* [46], 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.

3.2.3 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 indicate 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 that 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 Figure 3.1b, 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 Figure 3.1a, 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 Figure 3.5. 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.

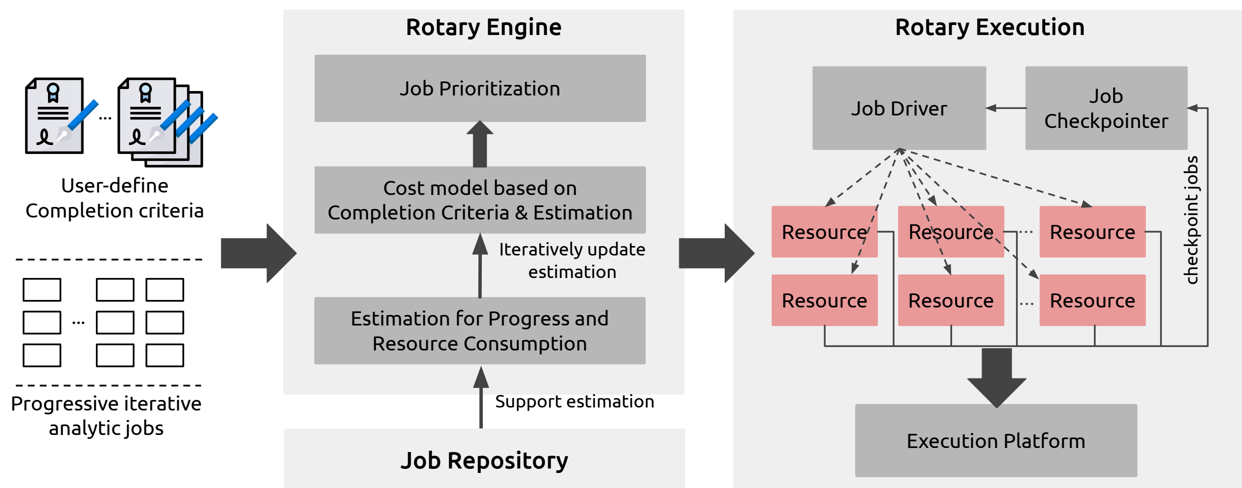


Figure 3.5: Framework architecture of Rotary

Rotary can also 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.

3.2.4 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 ith 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 bipartite 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 check-pointed 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 the completion criteria of all the jobs currently in the queue.

$$\pi : Q_t \mapsto \texttt{assign}(W, M) \tag{3.1}$$

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.

3.2.5 Resource Arbitration Algorithm

We propose an algorithm sketch for addressing the problem, as presented in Algorithm 3. 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 3: Algorithm Sketch for Resource Arbitration

while *not all jobs reach completion criteria* **do**
- **for** *jobs is active but not attained* $j_i, i \leftarrow 1$ **to** n **do**
  - Estimate the attainment progress $\hat{\phi}_{j_i}$ for next epoch;
- Resource arbitration for active jobs based on $\{\hat{\phi}_{j_i} | \forall i = 1..n\}$;
- **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 (§3.3.1) and DLT (§3.3.2).

3.3 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.

3.3.1 Rotary-AQP Implementation

To implement Rotary-AQP, we modify a single-user progressive query processing system based on Apache Spark [10] 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 [174, 144, 3, 141]) 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 [56]. 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 [89]. Rotary-AQP keeps tracking the least and largest aggregation results within a time window (e.g., t epochs) and uses this gap to determine conver-

gence[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 3.5, 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 [25]. The real-time data can be conveniently obtained since Rotary-AQP tracks the running AQP jobs. We further exploit weighted linear regression [70] 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

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

Algorithm 4: 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

for *job* $j_i \in W$ *that arrives* **do**
  Mark job j_i as active and place it to the active queue AQ;

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, $D = D - 1$;

and query plans in the AQP, which has been well-studied. In our implementation, we exploit Apache Spark's CBO [31] 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 4. 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.

3.3.2 Rotary-DLT Implementation

Rotary-DLT follows the architecture in Figure 3.5. 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 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 [97], 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 re-

gression, 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 §3.2.1, 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 GPUs 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 [97], 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.

Algorithm 5: 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 $\{M_1, \cdots, M_D\}$

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;

Following the problem statement (§3.2.4), we devise a threshold-based resource arbitration algorithm for DLT (Algorithm 5). 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 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 6.

Algorithm 6: 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** $\phi_i = \frac{e^*}{c_i}$;

3.4 Evaluation

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

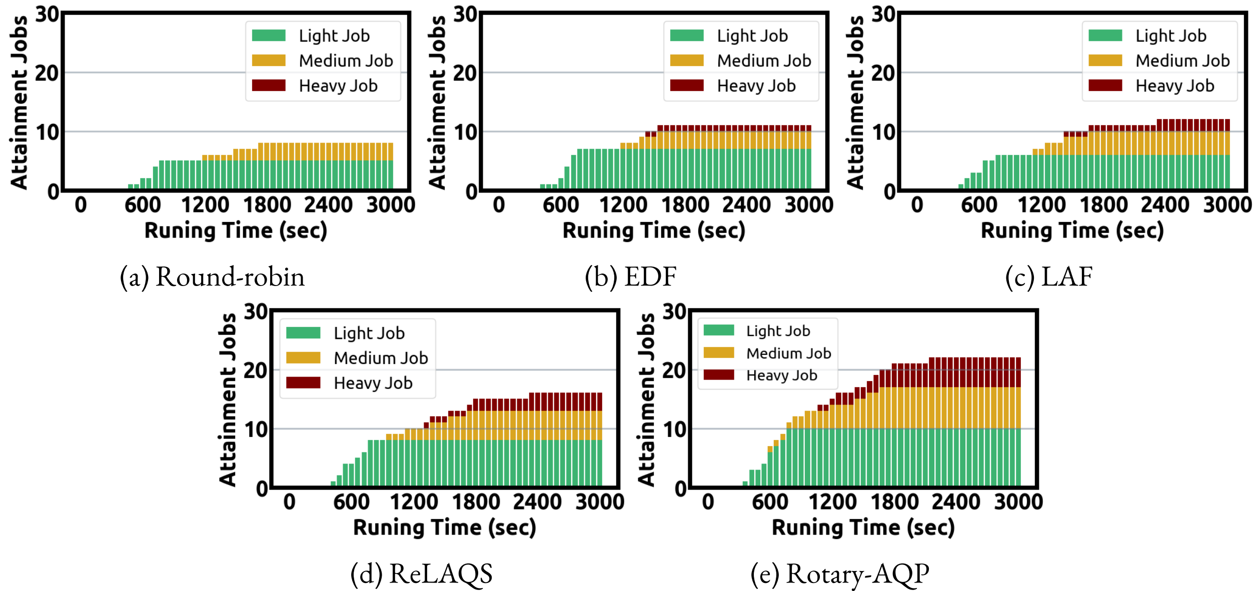
3.4.1 Rotary-AQP Evaluation

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? (§3.4.1)
- What is the overhead of resource arbitration? (§3.4.1)
- How does the distribution of job resource requirements impact the performance of Rotary-AQP? (§3.4.1)
- How does the progress estimation impact the performance of Rotary-AQP? (§3.4.1)

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 [8] cluster on a different machine with the same hardware configuration as the data source for AQP queries.

We implement four baselines for comparison: ReLAQS [144], 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.



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

Figure 3.6: Evaluation of Rotary-AQP and four baselines (Round-robin, EDF, LAF, ReLAQS) on the synthetic AQP workload

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 distribution with a mean arrival time of 160 seconds. The configurations of the workload are elaborated in Table 3.1.

<table>
<tr><td rowspan="3">Queries</td><td>Light</td><td>q1, q2, q4, q6, q10, q11, q12, q13, q14, q15, q16, q19, q22</td></tr>
<tr><td>Medium</td><td>q3, q5, q8, q17, q20</td></tr>
<tr><td>Heavy</td><td>q7, q9, q18, q21</td></tr>
<tr><td rowspan="2">Completion Criteria</td><td>Accuracy</td><td>55%, 60%, 65%, 70%, 75%, 80%, 85%, 90%, 95%</td></tr>
<tr><td>Deadline</td><td>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</td></tr>
<tr><td>Workload</td><td colspan="2">40% AQP jobs with light queries
30% AQP jobs with medium queries
30% AQP jobs with heavy queries</td></tr>
</table>

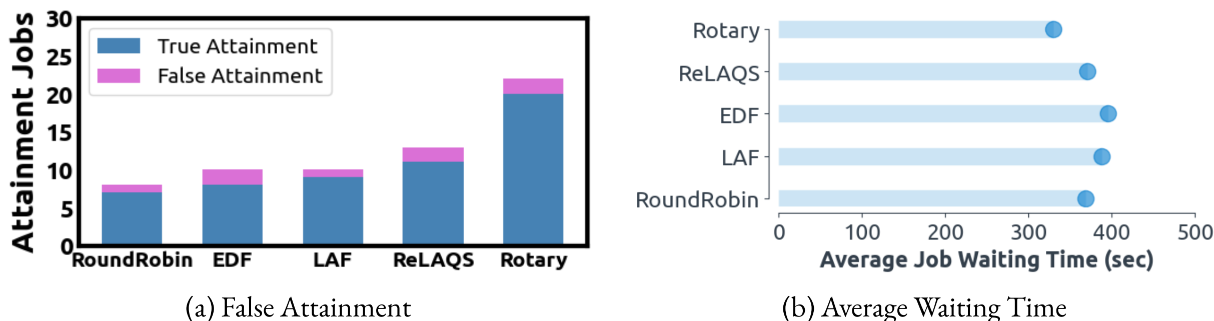
Table 3.1: Synthetic AQP workload. The selection of query type, accuracy threshold, and deadline are all random and based on a uniform distribution. Job arrival is based on a Poisson distribution.

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. Figure 3.6 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 of gaining more resources for running. Such results confirm the efficiency and effectiveness of Rotary-AQP.

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 that 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 Figure 3.7a. 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.



(a) False Attainment

(b) Average Waiting Time

Figure 3.7: False attainment and waiting time of Rotary-AQP

We also tally the average waiting time of the jobs in the workload, as shown in Figure 3.7b. 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.

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 Figure 3.8, 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.

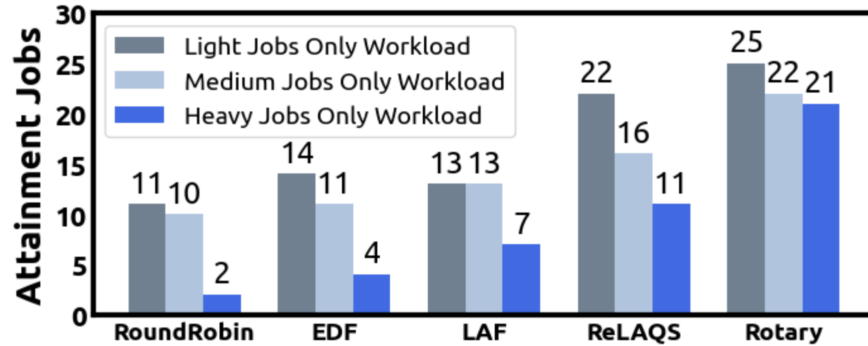
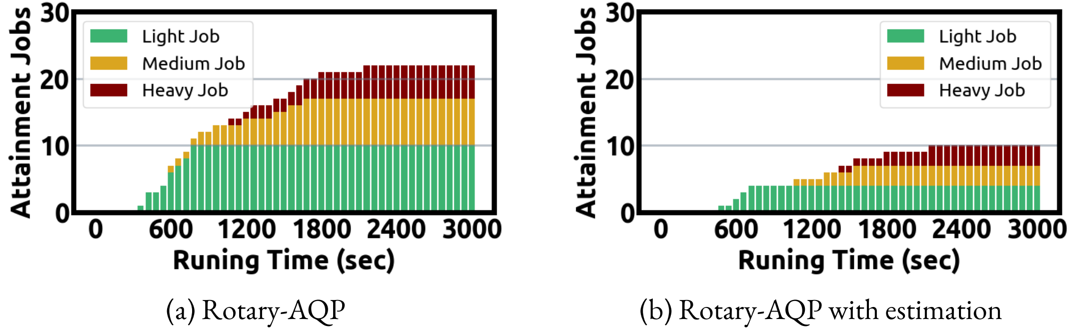


Figure 3.8: Attained jobs in the various workloads (30 jobs)

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.

Figure 3.9b displays the number of attained jobs under such artificial estimation, which is slightly better than round-robin (Figure 3.6a) and almost tied to EDF (Figure 3.6b) and LAF (Figure 3.6c). 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

Figure 3.9: Impact of progress estimation

3.4.2 Rotary-DLT Evaluation

We implement Rotary-DLT on top of TensorFlow 1.15 [151]. 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.

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 3.2. 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 [109] but choose bigger sizes for NLP models due to common practice [15]. 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.

<table>
<tr><td rowspan="5">Model</td><td>Architecture</td><td>Inception[147], MobileNet[60], MobileNetV2[135],
SqueezeNet [63], ShuffleNet[178], ShuffleNetV2[103],
ResNet[54], ResNeXt[170], EfficientNet[149], LeNet[81],
VGG[142], AlexNet[77], ZFNet[173], DenseNet[61],
LSTM[58], Bi-LSTM[47], BERT[157]</td></tr>
<tr><td>Batch size</td><td>Computer vision models: 2, 4, 8, 16, 32 [109]
Natural language processing models: 32, 64, 128, 256</td></tr>
<tr><td>Optimizer</td><td>SGD, Adam, Adagrad, Momentum</td></tr>
<tr><td>Learning rate</td><td>0.1, 0.01, 0.001, 0.0001, 0.00001</td></tr>
<tr><td>Dataset</td><td>Computer vision models: CIFAR-10 [76]
Natural language processing models:
UD Treebank [158], Large Movie Review Dataset [105]</td></tr>
<tr><td rowspan="4">Completion Criteria</td><td>Convergence-oriented criteria (delta accuracy)</td><td>5%, 3%, 1%, 0.5%, 0.3%, 0.1%, 0.05%, 0.03%, 0.01%, 0.005%, 0.003%, 0.001%</td></tr>
<tr><td>Accuracy-oriented criteria (final accuracy)</td><td>70%, 72%, 74%, 76%, 78%, 80%, 82%, 84%, 86%, 88%, 90%, 92%</td></tr>
<tr><td>Runtime-oriented criteria (epoch)</td><td>From scratch 5, 10, 30, 50, 100
Pre-trained (Fine-tuned): 1, 2, 3, 4, 5</td></tr>
<tr><td>Maximum epoch for criteria</td><td>1, 5, 10, 15, 20, 25, 30</td></tr>
<tr><td>Workload</td><td>Synthetic workload</td><td>60% DLT jobs with convergence-oriented completion criteria
20% DLT jobs with accuracy-oriented completion criteria
20% DLT jobs with runtime-oriented completion criteria</td></tr>
</table>

Table 3.2: Synthetic DLT workload. The selection of model architecture and proportion of jobs with various completion criteria distribution is based on the responses to our survey, and the selection of other hyperparameters and the parameters about completion criteria follow the uniform distribution.

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 $\frac{\textit{current accuracy}}{\textit{completion criteria}}$. 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 $\frac{\textit{current epoch}}{\textit{convergence-line}}$. For the jobs that failed to converge, we use $\frac{\textit{current epoch}}{\textit{max epochs}}$ instead.

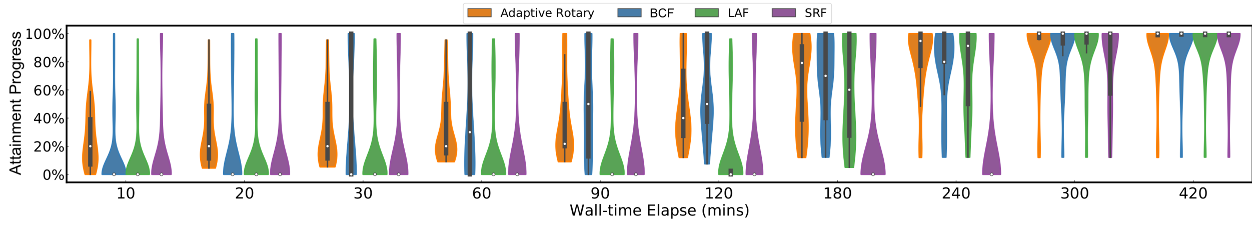
- *Runtime-oriented attainment progress*: The runtime-oriented attainment progress is denoted as $\frac{\textit{current epoch}}{\textit{completion criteria}}$, which is further exemplified by the following case. If a job has a runtime-oriented completion criterion of 15 epochs, and the attainment progress is $\frac{5}{15} = 33.3\%$ after training 5 epochs.

We evaluate the Rotary-DLT against three baselines:

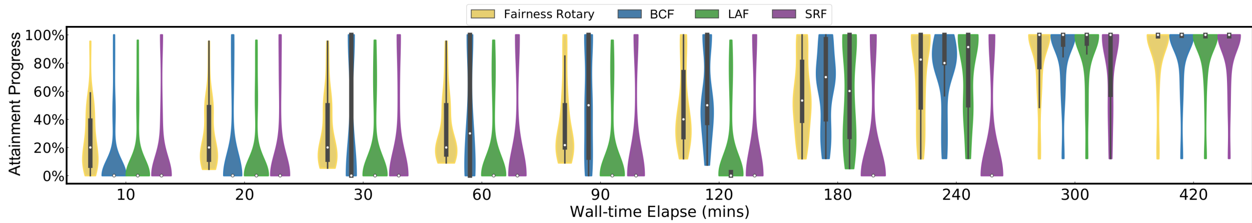
(a) Shortest Runtime First (SRF): it always runs the jobs with the shortest runtime completion criteria first and handles the other jobs following a round-robin strategy.

(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.

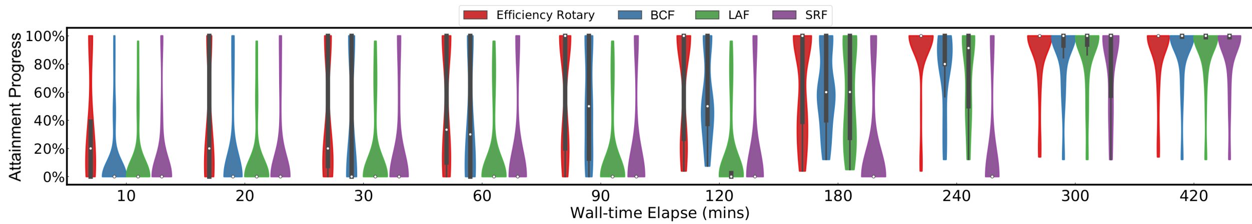
(c) Lowest Accuracy First (LAF): it always runs the jobs with the lowest accuracy completion criteria first and handles the other jobs following a round-robin strategy.



(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.

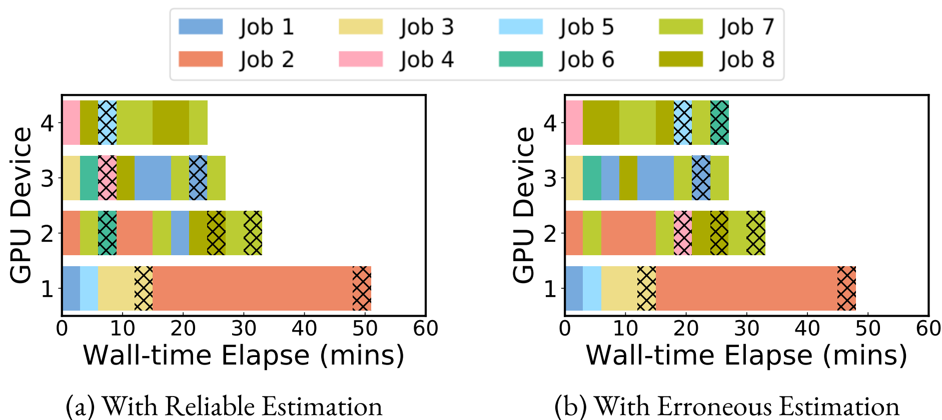
Figure 3.10: Evaluation of Rotary-DLT variants and three baselines on the synthetic DLT workload

We demonstrate all the results in Figure 3.10 using violin plots. In Figure 3.10a, 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. Figure 3.10b and 3.10c 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.

Impact of Training Epoch Estimation

Training epoch estimation is positioned at a vital place in developing and evaluating Rotary, and it is critical to understand its effect. 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, 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).



(a) With Reliable Estimation

(b) With Erroneous Estimation

Figure 3.11: Job placements under efficiency Rotary-DLT.

We demonstrate the placements for eight jobs in Figure 3.11. Each rectangle denotes a job placement, and the one with hatches means the job meets the completion criteria. Figure 3.11a 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 Figure 3.11b, 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.

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 3.3, 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 3.3: The overall process time and overhead in Rotary

3.5 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 allow 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 [1]. 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.

CHAPTER 4

RIVETER: ADAPTIVE QUERY SUSPENSION AND RESUMPTION FRAMEWORK FOR CLOUD NATIVE DATABASES

With the recent advancement in modern hardware and virtualization, an increasing number of data processing and analytic workloads are shifting towards database systems deployed on large-scale cloud infrastructure [131]. This shift is giving rise to the emergence of cloud-native databases [163]. Nevertheless, the increasing prevalence of ephemeral cloud resources is prompting a re-thinking of the principles of cloud-native databases and necessitating a novel query execution. First, *ephemeral cloud resources are ever-changing in availability*. Spot instances [6, 177, 29], which provide short-lived computing infrastructure for short-running jobs. Recent developments in cloud computing are amplifying this transient capacity. One noteworthy example is the rise of serverless computing [140, 127], which empowers applications to leverage lightweight cloud resources characterized by constrained runtime, memory capacity, and computation specification. Another emerging trend is "zero-carbon clouds" [28]. In this cloud paradigm, data centers are designed to be entirely ephemeral, driven primarily by renewable energy sources, such as sunlight and wind, which are inherently unstable in availability. Second, *the monetary cost of cloud resources can be fluctuating*. The inherent multi-tenancy nature of cloud resources [118] inevitably leads to price adjustments based on resource supply and demand dynamics. [7]. Reportedly, the prices of cloud resources can surge to 200 to 400 times the normal rate during peak demand [140]. An increasing number of users are leaning towards cost-effective choices, even if they entail slightly higher latency or potentially outdated results [4]. Moreover, the recent rise of data science and data-driven AI has introduced increasingly complex and heterogeneous workloads, combining both long-running and short-running queries [100, 44, 96]. This diversity can potentially lead to resource saturation, thereby exacerbating the fluctuations in pricing.

These two emerging trends challenge the cloud-native databases from two perspectives: (1) the assumption that cloud resources are stable is not applicable, and thus preserving the resources for a relatively long term is not always feasible; (2) the widely-used latency-oriented SLA (Service Level Agree-

ment) may not be the best option for all users, particularly for those who prioritize cost-efficiency over speed. An ideal cloud-native database designed for ephemeral resources needs to perform queries in an adaptive manner: *suspending a query when resource accessibility or cost-efficiency is no longer viable and subsequently resuming the query when resources become available or cost-effective again while minimizing the potential overhead caused by the query suspension and resumption.*

Motivated by this novel requirement, we proposed *Riveter*, an adaptive query suspension and resumption framework. Riveter supports various query suspension and resumption strategies, including the redo, process-level, and pipeline-level strategy. Specifically, the redo strategy allows for terminating a query at any given moment and subsequently rerunning it when necessary or advantageous. However, with this strategy, all current progress is forfeited upon termination. In contrast, the process-level strategy can also suspend a query at any point in time but preserve the current progress by persisting all intermediate data and context information of the process in which the query is executed. Nevertheless, it comes with notable drawbacks: the persisted data can be exceedingly large, and the requirement for resuming processes requires identical resource configurations, such as the number of hardware threads and allocated memory size, as were in use at the time of suspension. The pipeline-level strategy offers an alternative approach by persisting the intermediate data of each pipeline in the query plan for potential resumption. This implies that suspension and persistence of intermediate data only occur after specific pipelines have completed their execution, but this strategy usually significantly reduces the volume of data persisted since the intermediate data is typically aggregated upon pipeline completion. Riveter also employed a cost model that can estimate the latency of various strategies and thereby adaptively determine if, when, and how to suspend queries.

For the implementation of Riveter, we modify DuckDB [38] to realize the pipeline-level strategy and develop the process-level strategy building on top of CRIU (Checkpoint/Restore In Userspace) [32]. We also devise an adaptive query suspension and resumption algorithm based on the proposed cost model. We conducted a performance study using the TPC-H benchmark [154] to investigate the effectiveness of process-level and pipeline-level strategies. For instance, the process-level strategy suspends query 21 at

approximately 50% of the execution time and requires 28GB of intermediate data storage, whereas the pipeline-level strategy only holds 112KB of intermediate data by delaying the suspension and completing the current pipeline. We also demonstrate how Riveter determines the optimal strategy for selected queries through an end-to-end analysis, and our results reveal that the optimal strategy introduces negligible latency for query suspension when applied to SF-100 datasets. Additionally, we validate the efficiency of our cost model in terms of estimation accuracy and runtime performance through a cost model evaluation.

Contributions. The goal of this work is to propose and design an adaptive query execution framework for the evolving paradigms of cloud-native databases. Our contributions include:

- Characterizing the adaptive query execution based on suspension and resumption and exploring the opportunities for cloud-native databases (§4.1).
- Designing and implementing an adaptive query execution framework, Riveter, with supporting various query suspension and resumption strategies (§4.2.1, §4.2.2).
- Devising a cost model and an algorithm to adaptively determine the suspension and resumption strategies (§4.2.3).
- Conducting three evaluations for Riveter based on TPC-H benchmark (§4.3.1, §4.3.2, §4.3.3).

4.1 Motivation

In this section, we delve into the concepts of query suspension and resumption and further highlight the benefit and essential role of adaptive query suspension and resumption, particularly in cloud-native databases.

4.1.1 Query Suspension and Resumption

The concept of query suspension and resumption is rooted in recovery mechanisms in database systems [113, 106], and one of the most early work of query suspend and resume is database checkpoint mech-

anism which can create a point from which the execution engine can persist the current state of the database into non-volatile storage [50]. In light of the checkpoint mechanisms, a query suspend and resume approach is proposed for pull-based query execution [24]. It essentially creates a query suspend plan that may involve a combination of persisting current state and going back to previous checkpoints. Within this approach, the potential suspension points are always the ones that have minimized memory usage, thus mitigating significant overhead during suspension and resumption. Alternatively, instead of persisting the intermediate data, certain systems suspend queries and retain the intermediate data in memory during suspension. For instance, Amber [79] converts the operator DAG of the execution plan to an actor DAG and passes messages of ”suspend” and ”resume” among actors so that the query suspension or resumption can be triggered without persisting the intermediate data. Nevertheless, maintaining query processing in a suspended state without persistence can consume substantial memory resources and lead to significant delays for other queries. Moreover, this approach is not suitable for query or database migration which is not uncommon in cloud-native databases.

We identify three pivotal perspectives of suspension and resumption strategies: (1) suspension flexibility, (2) progress preservation, and (3) intermediate data persistence. The most ideal suspension and resumption strategy is the one that can suspend a query at any given time and preserve the progress of the suspending time point with the least intermediate data to persist. Suspension flexibility measures the granularity of triggering suspension a strategy can support. Within this spectrum, two extremes can be identified: one where queries can be suspended at any point in their execution, and another where suspension occurs only upon query completion. Progress preservation signifies the extent to which query processing progress can be retained, whether partially or in its entirety when a suspension is initiated. The intermediate data persistence specifically refers to the size of persisted intermediate data during suspension, which directly impacts the latency of query suspension.

We describe three strategies according to the above perspectives, as presented in Table 4.1.

Redo strategy allows for query suspension at any given time through immediate termination. It preserves no progress of query processing, resulting in no persisted intermediate data.

	Flexibility Granularity	*Persistence Granularity*	*Progress Granularity*
Redo Strategy	Terminate at anytime	No intermediate data	Lost all progress
Pipeline-Level Strategy	Suspend at pipeline completion	Intermediate data of pipeline	Keep progress of persisted pipeline
Process-Level Strategy	Suspend anytime at process level	Intermediate data of process	Preserved all progress

Table 4.1: Representative suspension & resumption strategies

Pipeline-level strategy only checks if a query can be suspended when a pipeline is completed and preserves progress by persisting the intermediate data at this completion point.

Process-level strategy can suspend a query at any given time and keep the current progress during suspension by persisting all necessary intermediate data.

4.1.2 Motivational Cases

Adaptive query suspension and resumption can play an essential role in the evolving scenarios of cloud-native databases, considering that cloud resources are becoming increasingly ephemeral in availability and fluctuating in monetary cost. The rise of spot instances [6, 177, 29] and the emergence of zero-carbon clouds [28, 148] have provided transient and intermittent computing infrastructures. Meanwhile, the price of these constrained cloud resources can escalate significantly when a substantial number of users compete for them during the peak period. In such scenarios, the importance of adaptive query execution becomes evident, allowing queries to run when resources are available or costs are within acceptable bounds and suspending them when resources become unavailable or costs become prohibitive.

We describe the following motivational cases in cloud-native databases for adaptive query suspension and resumption:

Case 1: Heterogeneous workloads. Cloud native databases are deployed for multi-tenants and process the heterogeneous workloads that are mixed by short-running and long-running queries. When the long-running queries monopolize resources for unexpectedly long periods, they potentially cause resource sat-

uration. This can lead to significant delays for shorter-running queries, which otherwise could have been completed promptly with sufficient resources. Existing methods, such as dynamic query scheduling or resource reservation, may fall short of expectations as they often treat the queries as indivisible entities, whereas the proposed adaptive query execution essentially converts a long-running query into a series of short-running ones by suspending and resuming it for multiple times, thereby allowing more flexible scheduling for execution.

Case 2: Database migration. Database migration is vital for maintaining elasticity and scalability in cloud-native databases. The current state-of-the-art approach is live database migration, which strives to move database services with minimal service disruption and negligible impact on performance. However, migrating entire database systems remains a non-trivial task and brings substantial overhead. Resource-adaptive query execution offers a promising solution by enabling the migration of individual queries, which can significantly reduce the overhead associated with migrating cloud-native databases. Furthermore, in certain circumstances where a single resource-intensive, long-running query dominates the database, migrating that specific query can free up resources for more efficient utilization.

Case 3: Computation with ephemeral resources. An increasing number of cloud-native databases now support serverless computing, which leverages lightweight and short-lived resources to enable users to invoke functions deployed in the cloud and retrieve results locally. Typically, these functions can be likened to queries or analytical tasks performed on specific datasets. However, users may encounter high latency when serverless computing resources are unstable or experience elevated costs, especially during peak usage times. This is attributed to the growing resource costs and unpredictable resource availability. The proposed resource-adaptive query execution can mitigate these issues by avoiding periods when resources are either unavailable or expensive for utilization.

4.2 Riveter Design and Implementation

Inspired by the motivational cases, we propose Riveter, an adaptive query suspension and resumption framework. Riveter supports three strategies as we identify in §4.1.1 and adaptively select the strategy

that is associated with minimized latency based on a cost model. The architecture of Riveter is described in Figure 4.1.

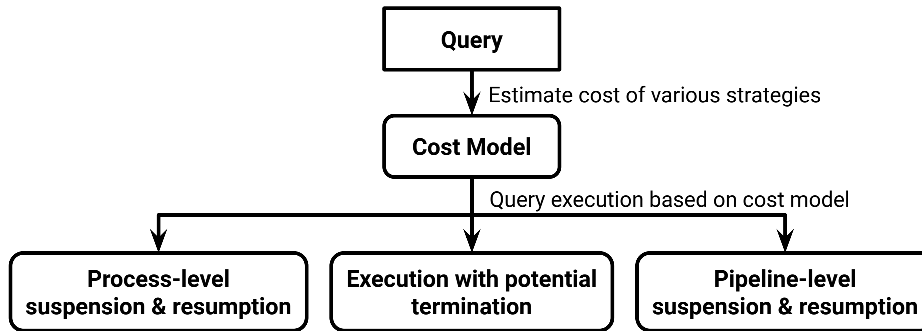


Figure 4.1: Riveter architecture

4.2.1 Pipeline-level Suspension and Resumption

Pipeline-based query execution divides the query plan into a number of pipelines for parallel and efficient execution. Dividing query plans into pipelines is based on the pipeline breakers that are usually blocking operators and collecting sufficient intermediate results for further process. Thus, the pipeline-level suspension and resumption strategy in Riveter takes the pipeline breakers as the natural suspension and resumption points.

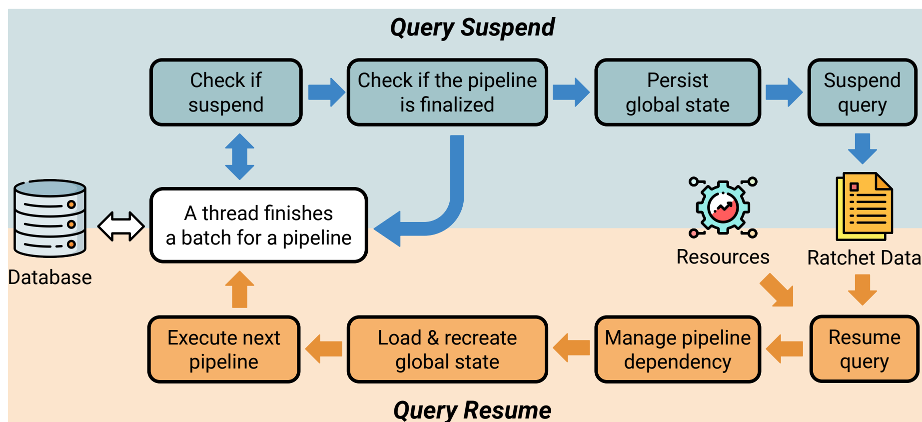


Figure 4.2: Workflow of pipeline-level query suspension and resumption strategy

To implement the pipeline-level strategy, we modified DuckDB [38] to implement a prototype system. The workflow of suspension and resumption is illustrated in Figure 4.2. When a thread completes

processing a pipeline with a data batch, Riveter checks whether the query execution has reached the suspension point and determines whether to initiate the suspension process. If affirmative, Riveter proceeds to verify if the current pipeline is finalized—indicating that all the intermediate results at the thread level have been merged into the global state. If this is the case, Riveter serializes and persists in the global state before suspending the query. When resuming a query, Riveter first attempts to identify the persisted intermediate data (i.e., *Ratchet Data* in Figure 4.2) and updated resource configurations in order to re-establish the query execution environment. Subsequently, Ratchet manages pipeline dependencies; for example, it bypasses pipelines processed prior to suspension and marks successor pipelines as ready for execution while also reconstructing the global states for future execution. Afterward, Riveter carries out the query until it reaches the next suspension point or produces the final results.

We illustrate this pipeline-level suspension and resumption strategy for some widely-used query operators in Figure 4.3. Although a pipeline can be executed by multiple threads, it maintains a global state for the intermediate data. Taking in-memory hash join as an example, the query plan is split into two pipelines: one is for building hash tables, and the other is for probing hash tables. Thus, there are two pipeline breakers at the end of each pipeline for suspension and resumption, and each breaker maintains a global state. The global state is persisted during suspensions.

Although the implementation of pipeline-level strategy only performs at different pipeline breakers for query suspension and resumption, the intermediate data are usually processed and aggregated, which makes the necessary persisted data relatively small. Our implementation brings an extra advantage, running queries using adaptive resources. This is because Riveter can check and exploit the available resources when loading and recreating the execution context for resume queries.

4.2.2 Process-level Suspension and Resumption

Riveter also provides the capability to effectively suspend and resume queries at any given point by implementing a process-level query suspension and resumption strategy. This strategy operates within the context of a process, ensuring the persistence of the complete context and all intermediate data upon ini-

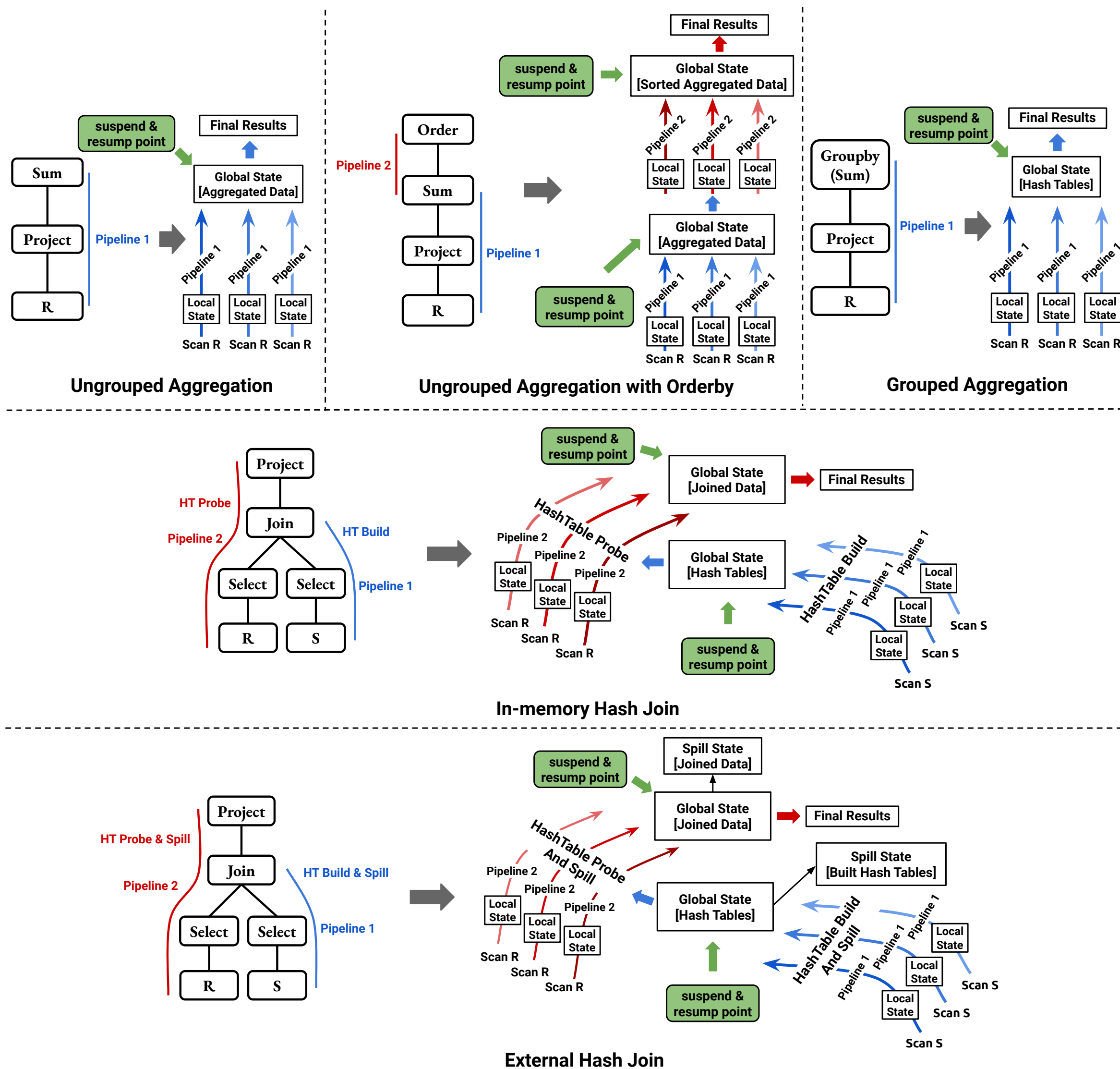


Figure 4.3: Pipeline-level suspension and resumption strategy for common operators in Riveter

tiation of suspension. Riveter reconstructs the entire process context and execution environment during the resumption phase, subsequently reloading the intermediary data associated with that process. Consequently, the process-level strategy in Riveter empowers the suspension and resumption of queries at will, albeit accompanied by a notable overhead due to the necessity of data persistence and loading. This overhead arises from the persistence of the entire process context and state during the suspension phase, which is then reinstated upon query resumption. Furthermore, the resumption of queries triggers the meticulous reconstruction of the process context to precisely match its state during suspension, which

also ensures that the resource configuration remains consistent and unaltered between the suspension and resumption phases.

We implement process-level query suspension and resumption in Riveter on top of CRIU tool [32], as demonstrated in Figure 4.4. When suspending a query, the entire query execution process will be dumped as multiple image files, which can be used to restore the process and the query for resumption.

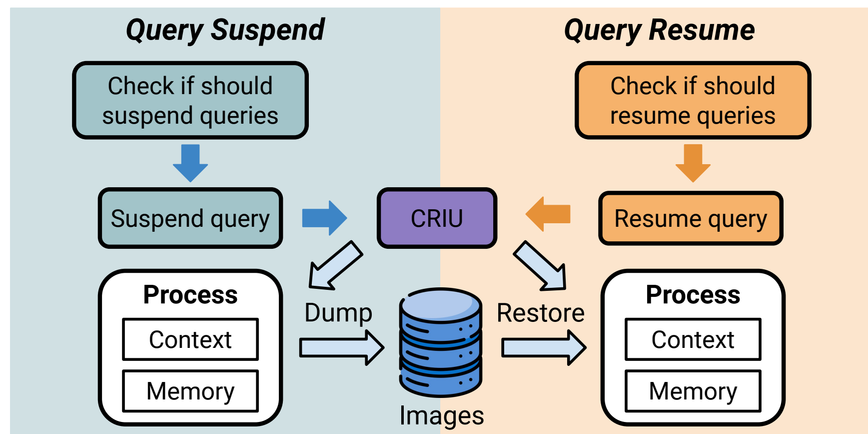


Figure 4.4: Workflow of process-level query suspension and resumption

4.2.3 Resource-adaptive Query Execution

We realize the adaptive query suspension and resumption in Riveter by supporting redo, pipeline-level, and process-level strategies and devise an associated cost model.

Cost Model

We devise a cost model to adaptively decide if, when, and how to suspend queries. We formulate the cost model as follows.

Given. A query q that may encounter a termination within a specific time window T, starting from T_s to T_e. Considering the potential termination, the query must confront a decision point: it can either be forcefully terminated, resulting in the loss of current progress, or strategically suspended while retaining all intermediate data. When the query is suspended before the termination, the intermediate data is per-

sisted, ensuring the continuity of the ongoing progress. However, persisting intermediate data at query suspension and reloading them during query resumption may introduce additional latency, denoted as L_s and L_r, which are denominated by the size of intermediate data. In cases where a query cannot be suspended before its intended termination, both the intermediate data and the current progress are forfeited, which necessitates a complete re-execution from its starting point.

Assumption. A termination can happen within a time window T, determined by a probability $P_T, 0 \leq P_T \leq 1$. Specifically, P_T can be either predefined or decided by a probability distribution π to simulate the termination in real-world applications. Additionally, the potential termination point $\pi(T)$ can differ each time the query is executed. The available amount of resources, such as CPU threads, memory, and I/O bandwidth, are known prior to the suspension and resumption of the query, but the availability of these resources might undergo changes during the suspension and resumption phases, thus, attempting to resume a query is unviable when the resources available are inadequate for its execution.

Objective. Select the strategy to minimize the overall cost for q. When executing query q with termination, the cost is,

$$Cost_q^{exec} = P_T^{redo} * \Gamma^{redo} \tag{4.1}$$

When executing query q with pipeline-level strategy, the cost is,

$$Cost_q^{ppl} = L_s^{ppl} + L_r^{ppl} + P_T^{ppl} * \Gamma^{ppl} \tag{4.2}$$

When executing query q with process-level strategy, the cost is,

$$Cost_q^{proc} = L_s^{proc} + L_r^{proc} + P_T^{proc} * \Gamma^{proc} \tag{4.3}$$

Specifically, $Cost_q^{exec}$ is the execution time before the termination point, which is jointly determined by the probability of termination P_T^{redo} and the redo cost Γ^{redo} under the redo strategy. The cost associated with pipeline-level and process-level strategies come from two perspectives: (1) latency caused by

persisting intermediate data during suspension and reloading them for query resumption, (2) the redo cost if the suspension fails to complete before the termination point. Therefore, the cost of pipeline-level and process-level strategies can be expressed as $Cost_q = (L_s + L_r) + P_T * \Gamma$. The latency L_s and L_r in pipeline-level and process-level strategies can be estimated based on the size of intermediate data and hardware specification to facilitate prompt decision-making during query execution.

Query Suspension and Resumption Algorithm

In accordance with the cost model, we have devised an algorithm for query suspension and resumption. Since continuously monitoring the status of query execution and performing cost calculations for potential suspension and resumption can significantly impede regular query execution, our algorithm employs a proactive approach to support adaptive query execution and meanwhile reduce the associated overhead.

Algorithm 7: Cost Estimation for Redo Strategy

Input : Query q, Termination time window $T[T_s, T_e]$, Probability of termination P_T

$T_{sum} = 0, N_{ppl} = 0$;
while *q reaches a pipeline breaker* **do**
  Observe the current time C_t;
  Observe the running time of the pipeline T_{ppl};
  $T_{sum} = T_{sum} + T_{ppl}, N_{ppl} = N_{ppl} + 1$;
  ======== *Cost Estimation for Redo Strategy* ========
  Initialize probability of termination using redo strategies P_T^{redo};
  if $C_t \geq T_s$ *or* $C_t + \frac{T_{sum}}{N_{ppl}} \geq T_e$ **then**
    $P_T^{redo} = P_T$;
  else if $T_s \leq C_t + \frac{T_{sum}}{N_{ppl}} < T_e$ **then**
    Overlapped time window $T_o = C_t + \frac{T_{sum}}{N_{ppl}} - T_s$;
    $P_T^{redo} = \frac{T_o}{T_e - T_s} * P_T$;
  else
    $P_T^{redo} = 0$;
  $Cost_q^{redo} = P_T^{redo} * T_e$;

Riveter estimates the cost of employing the three suspension strategies and makes latency-oriented decisions when the query execution reaches a pipeline breaker. For the query q, the cost of redo strategy $Cost_q^{redo}$ is expressed as $P_T^{redo} * \Gamma^{redo}$, where P_T^{redo} is calculated in terms of the overlap between ter-

mination time window T and the time of completing future pipelines. The completion time of future pipelines can be estimated based on the average of previous pipelines. The cost estimation for the redo strategy is illustrated in Algorithm 7.

Algorithm 8: Cost Estimation for Pipeline-level Strategy

Input : Query q, Termination time window $T[T_s, T_e]$, Probability of termination P_T

$T_{sum} = 0, N_{ppl} = 0$;

while q *reaches a pipeline breaker* **do**

  Observe the current time C_t;

  Observe current available memory M;

  Observe the running time of the pipeline T_{ppl};

  ======== *Cost Estimation for Pipeline-level Strategy* ========

  Initialize probability of termination using pipeline-level strategies P_T^{ppl};

  Obtain the size of intermediate data S^{ppl} of pipeline-level strategy;

  if $S^{ppl} \leq M$ **then**

    Estimate the latency L_s^{ppl} and L_r^{ppl} based on S^{ppl};

  else

    $L_s^{ppl} = \infty, L_r^{ppl} = \infty$;

  if $C_t + L_s^{ppl} \geq T_e$ **then**

    $P_T^{ppl} = P_T$;

  else if $T_s \leq C_t + L_s^{ppl} < T_e$ **then**

    Overlapped time window $T_o = C_t + L_s^{ppl} - T_s$;

    $P_T^{ppl} = \frac{T_o}{T_e - T_s} * P_T$;

  else

    $P_T^{ppl} = 0$;

  $Cost_q^{ppl} = L_s^{ppl} + L_r^{ppl} + P_T^{ppl} * C_t$;

For the cost estimation of pipeline-level strategy, we serialize and persist the intermediate data in binary format, essentially representing a large vector of bytes, which allows us to conveniently determine its size. Then, the latency L_s^{ppl} and L_r^{ppl} can be estimated based on the size of intermediate data and the random write/read speed. The P_T^{ppl} is also based on the overlap between the termination time window T and the estimated time of completing future pipelines, as shown in Algorithm 8.

The cost estimation of process-level strategy, as described in Algorithm 9, requires probing more future suspension points since the strategy can suspend queries anytime. The suspension point under a process-level strategy associated with minimum latency can be selected. During the probing process,

Algorithm 9: Cost Estimation for Process-level Strategy

Input : Query q, Termination time window $T[T_s, T_e]$, Probability of termination P_T

$T_{sum} = 0, N_{ppl} = 0;$
while *q reaches a pipeline breaker* **do**
  Observe the current time C_t;
  Observe current available memory M;
  Observe the running time of the pipeline T_{ppl};
  $T_{sum} = T_{sum} + T_{ppl}, N_{ppl} = N_{ppl} + 1;$
  ======== *Cost Estimation for Process-level Strategy* ========
  Initialize probability of termination using process-level strategies P_T^{proc};
  for st_i, $i \leftarrow C_t$ **to** $C_t + \frac{T_{sum}}{N_{ppl}}$ **do**
    Estimate the size of intermediate data $S_{st_i}^{proc}$ at st_i;
    if $S_{st_i}^{proc} \leq M$ **then**
      Estimate the latency L_{s,st_i}^{proc} and L_{r,st_i}^{proc} based on $S_{st_i}^{proc}$;
    else
      $L_{s,st_i}^{proc} = \infty, L_{r,st_i}^{proc} = \infty;$
    if $st_i + L_{s,st_i}^{proc} \geq T_e$ **then**
      $P_T^{proc} = P_T;$
    else if $T_s \leq st_i + L_{s,st_i}^{proc} < T_e$ **then**
      Overlapped time window $T_o = st_i + L_{s,st_i}^{ppl} - T_s$;
      $P_T^{proc} = \frac{T_o}{T_e - T_s} * P_T;$
    else
      $P_T^{proc} = 0;$
    $Cost_q^{proc} = \min\{L_{s,st_i}^{proc} + L_{r,st_i}^{proc} + P_T^{proc} * st_i\}$

Riveter employs an iterative approach, advancing suspension time points by each time unit that can be predefined according to the termination time window to estimate the latency of each potential suspension point in the near future. However, it is non-trivial to estimate the latency L_s^{proc} and L_r^{proc} of process-level strategy at each potential suspension point since the intermediate data include not only the processing data but also all the necessary context information. To address this challenge, we exploit a regression-based approach to estimate the size of intermediate data under the process-level strategy. The regression-based approach essentially fits a curve based on a number of key factors, such as the size and the cardinality of the input data, the metadata of the query (e.g., number of join and group-by), and the suspension point. The above estimations for the latency of pipeline-level and process-level strategies are practical for efficient decision-making processes in Riveter and can be improved or replaced with more

sophisticated methods in the framework.

After estimating the latency of the three strategies, Riveter selects the strategy with the least estimated latency. We present an illustrative example in Figure 4.5.

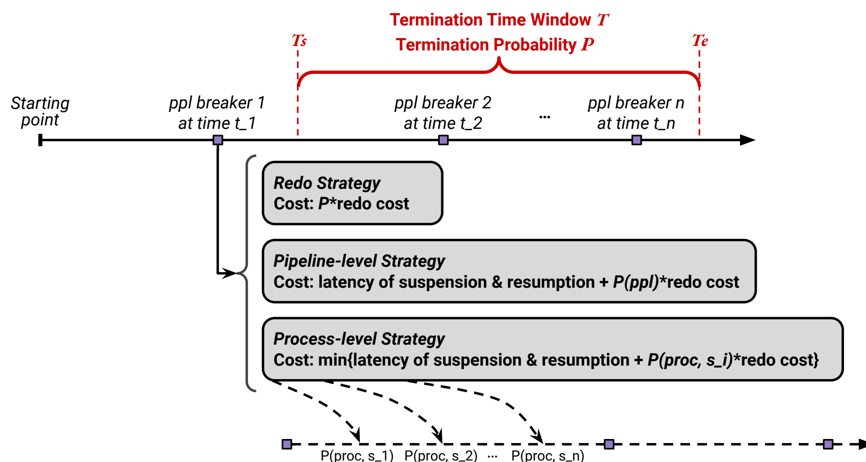


Figure 4.5: Illustrative example of query suspension and resumption algorithm

4.3 Evaluation

We conduct three evaluations for Riveter using TPC-H benchmark:

- Investigating the intermediate data persistence under process-level and pipeline-level strategy (§4.3.1).
- Analyzing the strategy selection based on the cost model (§4.3.2).
- Studying estimation accuracy and runtime of the cost model (§4.3.3).

Riveter is evaluated on a server with two Intel Xeon Silver CPUs (2.10GHz, 12 physical cores), 192GB memory, and 60TB hard disk (7200 RPM, SATA 6.0 Gb/s), running Ubuntu Server 18.04. Within the evaluation, we use the parquet format [9] and assume that raw data have been ingested for query processing. During the evaluation, query executions are initiated using Python client APIs, which are commonly employed in real-world applications. All the results in the evaluation are averaged over 3 independent runs.

We run all the queries from the TPC-H benchmark to conduct a comprehensive evaluation. Additionally, we highlight the queries Q1, Q3, Q17, and Q21, which feature diverse core operators and varying numbers of input tables. The queries are characterized in Table 4.2.

Query	**Core Operators**	**Tables**
Q1	1 groupby	1 table
Q3	1 groupby, 2 join	3 tables
Q17	1 join, 1 unionall	2 tables
Q21	1 groupby, 4 join, 1 outer join	4 tables

Table 4.2: Selected queries in TPC-H

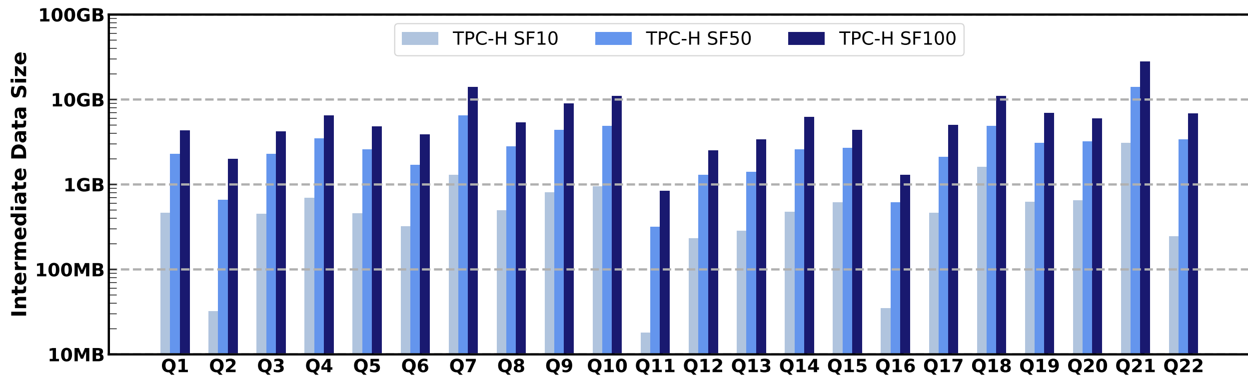
4.3.1 Impact of Intermediate Data Persistence during Suspension

In this evaluation, we investigate the size of persisted intermediate data of the strategies in Riveter. Since the redo strategy terminates queries and re-executes them subsequently without persisting any intermediate data, this evaluation focuses on the process-level and pipeline-level strategies employed in Riveter.

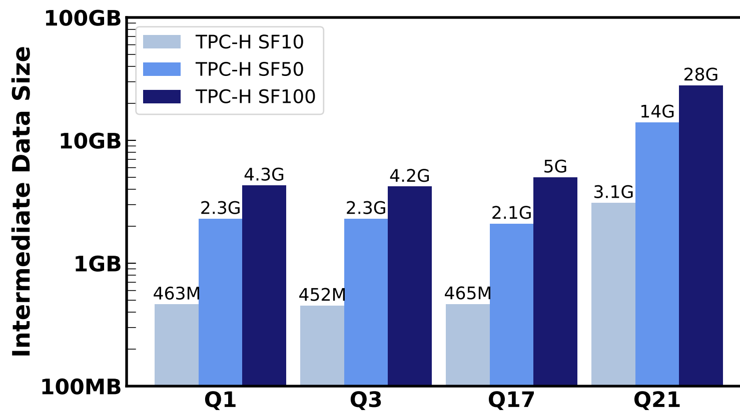
Process-level Strategy

We utilize the process-level strategy in Riveter to suspend all the queries from TPC-H benchmark. Specifically, Riveter suspends the queries at approximately 50% of their execution time, persisting all intermediate data and process context. As depicted in Figure 4.6a, the sizes of intermediate data for most queries exhibited a proportional increase with the volume of input datasets. Furthermore, Figure 4.6b provides a closer look at the selected queries: Q1, Q3, Q17, Q21. For example, when Q21 is suspended, the size of persisted data is 3GB for the SF-10 dataset, 14GB for the SF-50 dataset, and 28GB for the SF-100 dataset. However, there were exceptions noted for queries Q2, Q11, Q16, and Q22 when using the SF-10 dataset. This deviation can be attributed to the lightweight nature of these four queries, which complete rapidly when processing smaller datasets such as SF-10. For instance, the execution time of Q2 and Q11 are 0.9 and 0.4 seconds, respectively. Consequently, when Riveter suspends these queries at approximately 50%

of their execution time (i.e., around 0.45 and 0.2 seconds, respectively), they were still in the very early stages of execution, resulting in relatively small sizes of intermediate data.



(a) Persisted intermediate data size of all queries in TPC-H



(b) Q1, Q3, Q17, and Q21 of Figure 4.6a

Figure 4.6: Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the process-level strategy on SF-10, SF-50, SF-100 datasets

Following the above argument that queries regarding queries in their early execution stages generate fewer intermediate data under the process-level strategy, we further investigate how different suspension points affect the size of intermediate data. Specifically, we measured the intermediate data size when suspending queries at approximately 30%, 60%, and 90% of their execution progress. Figure 4.7 presents that the size of persisted intermediate data under the process-level strategy increases as suspension occurs later in the query execution. This trend is typically due to memory allocation persisting without timely deallocation during query execution. Consequently, intermediate data accumulates until the query execution

is completed. This observation underscores the trade-off between suspending a query immediately to reduce the space required for persistence and delaying suspension to make more progress, should it prove advantageous.

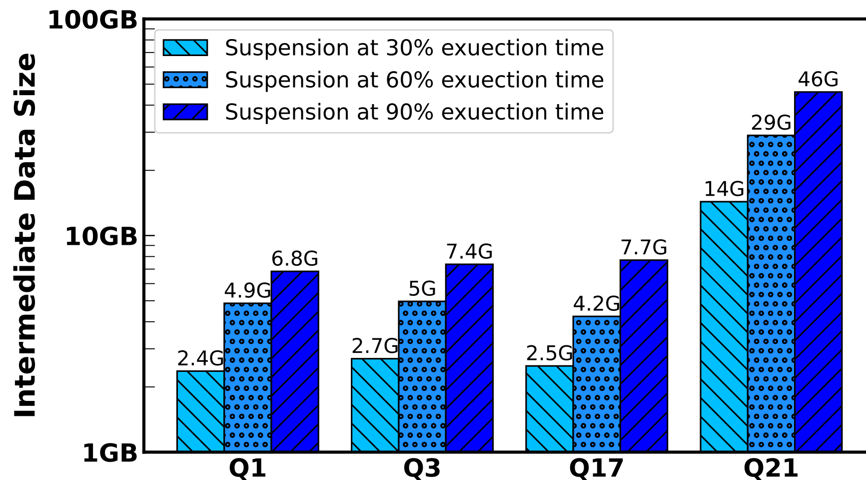
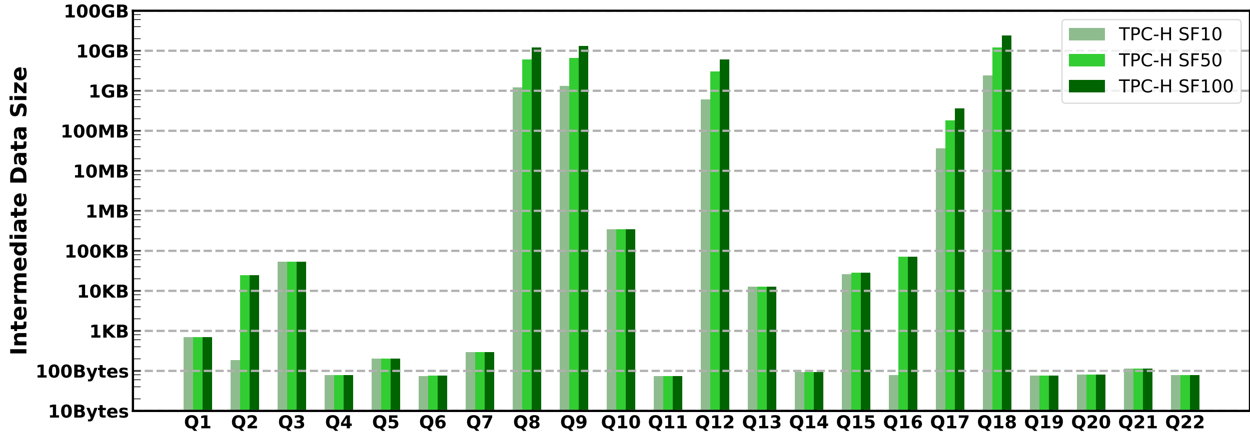


Figure 4.7: Persisted intermediate data size of queries (Q1, Q3, Q17, Q21) in TPC-H when suspending them at around 30%, 60%, and 90% execution time using process-level strategy on SF-100 dataset

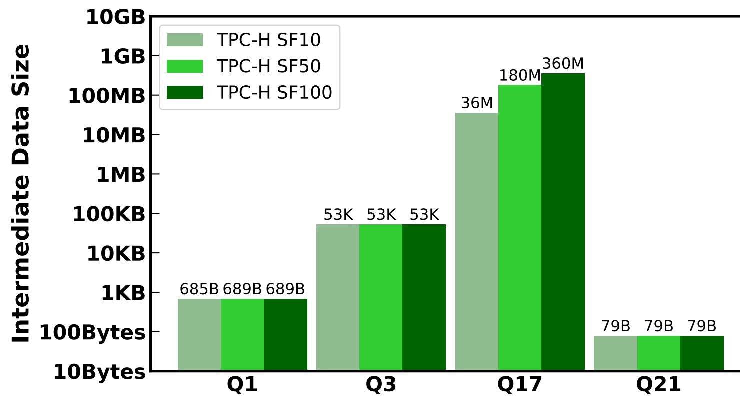
Pipeline-level Strategy

For consistency in the performance study, we exploit Riveter to suspend queries from TPC-H benchmark at around 50% of their execution time but using pipeline-level strategy. Since the strategy only suspends a query when one of its pipelines is finalized, the time a suspension is requested is not always the time the suspension process actually starts.

Figure 4.8a illustrates the size of persisted intermediate data of 22 queries, each exhibiting notable differences. We also provide specific numbers of selected queries in Figure 4.8b. For example, when suspending Q1 using the SF-10 dataset, the size of intermediate data is less than 1KB, while the intermediate data generated by Q8 for the same dataset is approximately 12GB. Furthermore, while the intermediate data for certain queries (e.g., Q8, Q9, Q12, Q17, and Q18) that undergo suspension tends to grow proportionally with the size of the input datasets (SF-10, SF-50, SF-100), the size of intermediate data for other queries, such as Q1, Q4-7, Q11, Q14, and Q19-Q22, remains consistent across these datasets. This



(a) Persisted intermediate data size of all queries in TPC-H



(b) Q1, Q3, Q17, and Q21 of Figure 4.8a

Figure 4.8: Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the pipeline-level strategy on SF-10, SF-50, SF-100 datasets

behavior is primarily influenced by the specific pipeline in which the query is situated when suspension is initiated. For instance, queries within hash-join pipelines tend to exhibit increased intermediate data sizes in tandem with larger input datasets, whereas queries within aggregation pipelines, such as those involving sum or average operators, tend to produce intermediate data of similar sizes due to the nature of their aggregated results. Moreover, in the case of certain queries, the pipeline-level strategy results in a greater volume of intermediate data compared to the process-level strategy. To illustrate, consider Query Q8: this discrepancy arises from the fact that the pipeline-level strategy defers the persistence of intermediate data until the pipeline engages with a hash-join operator is completed, thereby necessitating the retention of the entire hash table for the join. However, the process-level strategy exhibits a swifter re-

sponse to suspension requests, resulting in the persistence of intermediate data involving only the partial hash table.

Given the characteristics of the pipeline-level strategy, we measure the time difference between when the suspension is requested and when it actually occurs (i.e., the moment when the suspension process commences), as illustrated in Figure 4.9. Specifically, the Q21 has the minimum delay on different datasets due to the granularity of its query plan, which can be quantified by the number of pipelines involved. This is mainly because Q21 has many more pipelines than Q1, Q3, and Q17, which provides more feasible suspension points and thereby allows the query to be suspended closer to the suspension point. Hence, the performance study quantitatively confirms that when suspension occurs, the pipeline-level strategy typically generates significantly less intermediate data for persistence compared to the process-level strategy. This reduced intermediate data can be attributed to the fact that pipeline-level strategy does not necessitate capturing all state and context data for resumption. However, the pipeline-level strategy may introduce extra delays in responding to suspension requests, particularly when the query plan comprises fewer pipelines or is dominated by a single heavyweight pipeline.

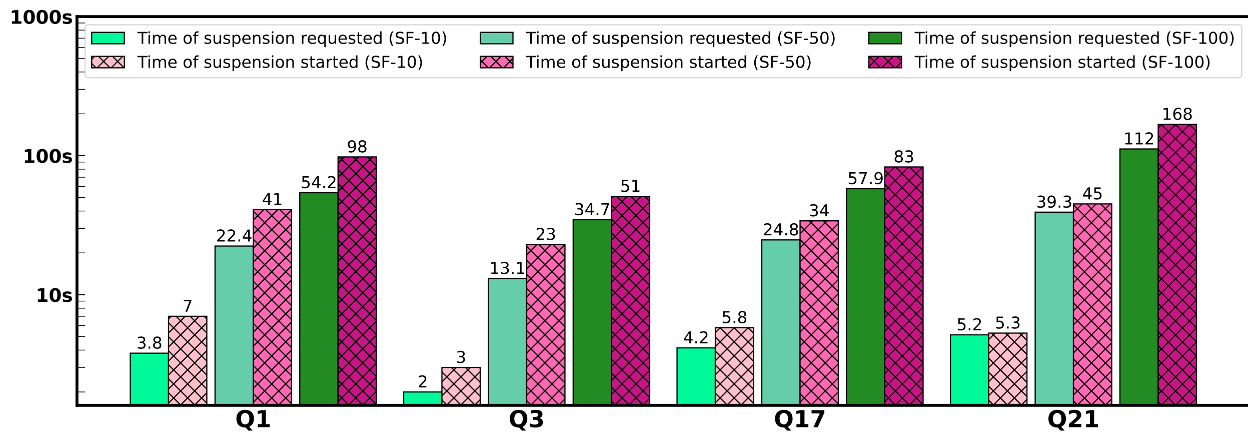


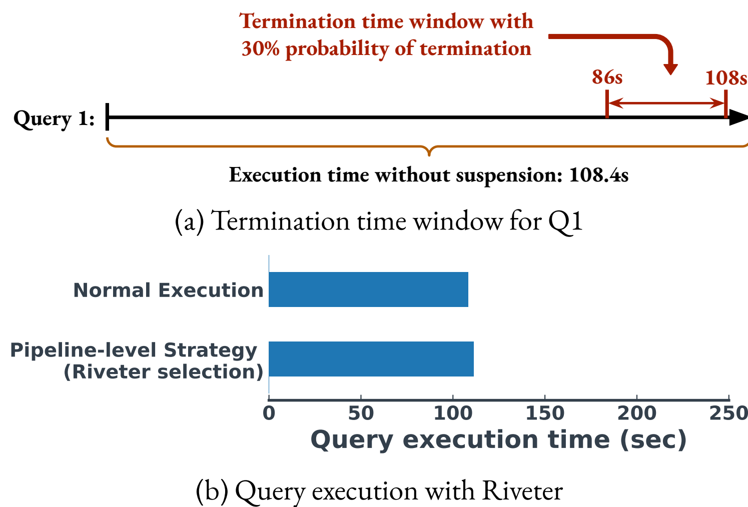
Figure 4.9: Time lag incurred when suspensions are requested under the pipeline-level strategy

Hence, the performance study quantitatively confirms that when suspension occurs, the pipeline-level strategy typically generates significantly less intermediate data for persistence compared to the process-level strategy. This reduced intermediate data can be attributed to the fact that pipeline-level strategy does not necessitate capturing all state and context data for resumption. However, the pipeline-

level strategy may introduce extra delays in responding to suspension requests, particularly when the query plan comprises fewer pipelines or is dominated by a single heavyweight pipeline.

4.3.2 Analysis of Suspension and Resumption Strategy Selection

We investigate Riveter's selection of suspension and resumption strategies for Q1, Q3, Q17, and Q21 by examining scenarios in which termination time windows and termination probabilities are configured with different settings. Specifically, the termination time window $[T_s, T_e]$ with termination probability P signifies that the query execution has a probability of P for being terminated while the execution is within the time window $[T_s, T_e]$. Specifically, each termination time point within the time window is assigned with equal termination probability P. We further compare query execution times under two conditions: without suspension, and with a single suspension and resumption following the strategy selected by Riveter. The total query execution time accounts for termination skipping, which can be achieved by straightforwardly incorporating termination time. We run the queries Q1, Q3, Q17, and Q21 under SF-100 datasets.



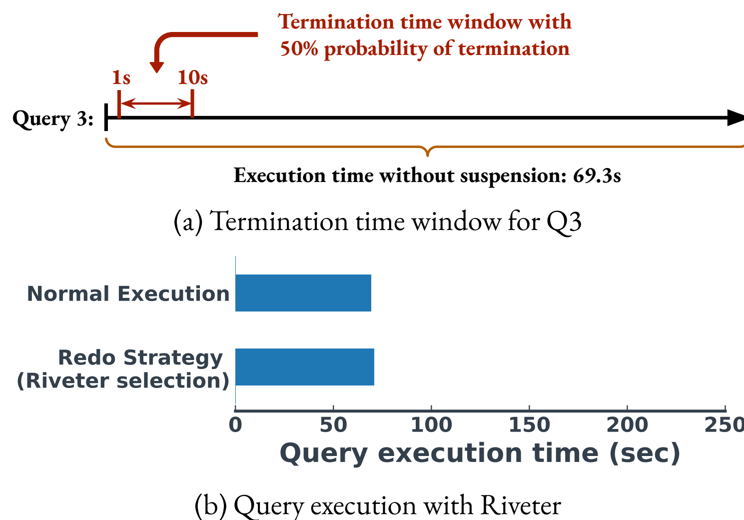
(a) Termination time window for Q1

(b) Query execution with Riveter

Figure 4.10: Analysis of strategy selection for Q1

Query Q1 serves as a representative example of queries that heavily rely on one of the few pipelines. For Q1, we configure a suspension time window $[86s, 108s]$, with a 30% probability of termination, as

presented in Figure 4.10a. Once Q1 approaches this time window, Riveter assesses the cost associated with different suspension strategies. The cost of the redo strategy is notably high in this scenario. This is primarily because Q1 is nearing the completion of its execution, and any termination at this point would result in the loss of all progress, necessitating a complete restart. Consequently, this would lead to a significant increase in latency. Meanwhile, the estimated cost of the process-level strategy is considerably higher than that of the pipeline-level strategy. This is attributed to the much larger size of intermediate data that must be persisted during the suspension period. Thus, Riveter recommends the pipeline-level strategy for query Q1, and the overall execution time under the pipeline-level strategy is approximately $111.4s$ compared with the normal execution time $108.4s$, shown in Figure 4.10b.



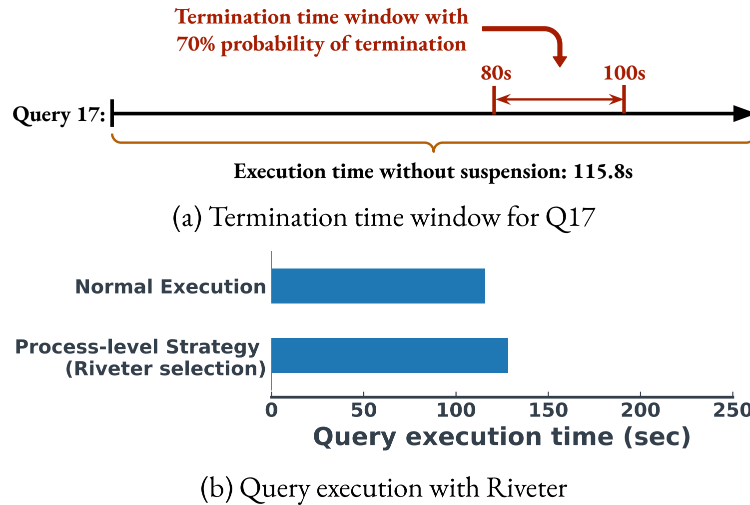
(a) Termination time window for Q3

(b) Query execution with Riveter

Figure 4.11: Analysis of strategy selection for Q3

In the case of Query Q3, as shown in Figure 4.11a, we examine Riveter's ability to identify the optimal suspension strategy when there is a potential for early termination during query execution. In this case, the first pipeline of Q3 reaches completion quite early in the process. Riveter, after performing its calculations, selects the redo strategy for suspension, primarily due to its lower cost compared to the other two strategies. This choice is bolstered by the fact that the termination time window occurs at the outset of query execution, resulting in a relatively modest cost for the redo strategy since the cost of the redo strategy is mainly associated with the query re-execution instead of persisting any intermediate data.

Thus, the query execution times under the redo strategy closely resemble those of normal execution. (Figure 4.11b).



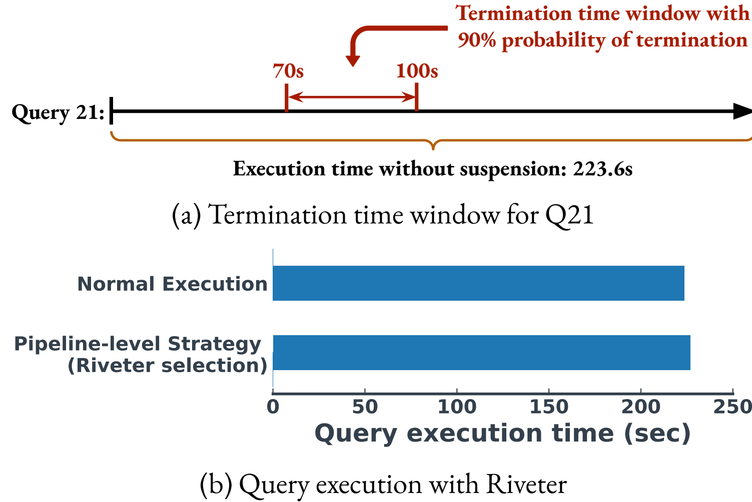
(a) Termination time window for Q17

(b) Query execution with Riveter

Figure 4.12: Analysis of strategy selection for Q17

In the case of Query Q17, as illustrated in Figure 4.12a, the probability of query termination is relatively high. Consequently, suspending the query becomes a more reasonable choice than redoing it entirely. Riveter, taking this into account, opts for the process-level strategy over the pipeline-level strategy. The latter, which involves persisting relatively large amounts of data, introduces additional latency. Furthermore, Riveter's estimation suggests that the next pipeline of Q3 will cover the termination time window, rendering the redo strategy more expensive due to the high termination probability. However, as depicted in Figure 4.12b, the query execution time under the process-level strategy is longer compared to execution without suspension. This is primarily because the cost model requires more time for estimation in this case.

Query Q21 is one of the most complex queries in TPC-H benchmark. We configure a termination time window at the middle phase of query execution, paired with a high termination probability, as illustrated in Figure 4.13a. According to Riveter's estimates, the size of intermediate data maintained for persistence during suspension, using a process-level strategy, substantially exceeds that of the pipeline-level strategy, which can introduce considerable latency into the query execution process. Additionally, the termination time window initiates at 70 seconds. This factor diminishes the attractiveness of employ-



(a) Termination time window for Q21

(b) Query execution with Riveter

Figure 4.13: Analysis of strategy selection for Q21

ing the redo strategy, given that termination has a 90% probability of occurring, effectively resetting the progress and significantly extending the overall query execution time. Therefore, Riveter recommends adopting the pipeline-level strategy, as the execution time with a single suspension and resumption event introduces only negligible latency (Figure 4.13b).

4.3.3 *Accuracy Estimation and Runtime of Cost Model*

The cost model plays a crucial role in Riveter framework, aiding in determining the optimal strategy based on latency estimations for various strategies. As detailed in § 4.2.3, assessing the latency of the process-level strategy necessitates estimating the size of intermediate data. Consequently, we conduct an investigation into the accuracy of this estimation within Riveter. Thus, we collect data from 200 query executions and employ a regression-based approach to fit the curve. Subsequently, we leverage this curve to estimate intermediate data size when applying the process-level strategy in Riveter to suspended queries. The results of our estimations are presented in Table 4.3, offering valuable insights to inform the decision-making process in the cost model and to identify the optimal strategy.

The regression-based estimation method demonstrates efficient performance, introducing negligible additional overhead during query execution. In our implementation of Riveter, the cost model is

Query	Dataset	Estimate	Ground-truth	Accuracy
Q1	SF-50	2.25GB	2.3GB	97.8%
Q1	SF-100	4.75GB	4.3GB	89.5%
Q3	SF-50	3GB	2.3GB	69.5%
Q3	SF-100	5.2GB	4.2GB	76.2%
Q17	SF-50	2.7GB	2.1GB	71.4%
Q17	SF-100	4.5GB	5GB	90%
Q21	SF-50	11.7GB	14GB	83.6%
Q21	SF-100	24.3GB	28GB	86.8%

Table 4.3: Estimation analysis of cost model when queries are suspended at around 50% using process-level strategy

launched as a dedicated process and efficiently communicates with the process where the query is executed through allocated shared memory. We measure the running time of the cost model, including data ingestion for estimation, latency estimation for the process-level strategy, and evaluation of latency for other strategies to determine the optimal strategy. Specifically, Riveter runs query Q1, Q3, Q17, and Q21 on the SF-100 dataset and suspends them at around 50% of their execution time. We capture the running time of the cost model when it is triggered for optimal strategy selection for this suspension request. As tabulated in Table 4.4, the running time of the cost model for Q1, Q17, and Q21 is negligible. The cost model for Q17 brings relatively higher overhead to the query execution, which is due to obtaining the size of intermediate data for further estimation.

Query	Running Time of Cost Model	Overall Execution Time (no suspension)
Q1	0.012s	108.4s
Q3	0.018s	69.3s
Q17	5.85s	115.8s
Q21	0.02s	223.6s

Table 4.4: Running time of cost model in Riveter based on the queries using SF-100 dataset

4.4 Related Work

In this section, we identify some key domains where Riveter can effectively contribute, and compare it with established relevant techniques within those domains.

4.4.1 Database Migration

Database migration is a service that facilitates the transfer of users' databases and analytics workloads from one execution environment to another. This process inevitably involves stopping and restarting the database, akin to suspending and resuming queries. One state-of-the-art method for database migration is known as live database migration. This technique, rooted in virtualization [30], aims to migrate database services with minimal service interruption and negligible performance impact [37, 167].

Live database migration encompasses two primary approaches: iterative copying [37, 35] and dual execution [41, 69]. In the iterative copying approach, akin to the methodology employed by Riveter, active transactions on the source database are temporarily suspended during migration. The migration process then iteratively transfers both the database cache and the state of ongoing transactions, allowing the migration destination to initiate with a hot cache [36, 166]. Conversely, dual execution, proposed in Zephyr [41] and refined in systems like MgCrab [95] and Remus [69], enables simultaneous execution of transactions on both the source and destination databases. This simultaneous execution minimizes service disruption and system downtime, ensuring service disruption and system downtime are unobtrusive. ProRea [136], on the other hand, seeks to amalgamate the strengths of iterative copying and dual execution. It initiates migration by proactively transferring the database cache and subsequently ensures that newly arriving transactions are processed concurrently at both the source and destination databases. In contrast to the aforementioned approaches, Slacker [14] and Madeus [112] represent middleware-based solutions that avoid direct modification of existing database systems. Slacker utilizes transaction logs to synchronize the destination database with the source, while Madeus concurrently propagates commit operations as well as the first read and write operations from the source to the destination during migration. However, despite their simplicity, these middleware-based methods provide limited performance

advantages when compared to solutions integrated into the core of the database management systems (DBMSs).

State migration plays a pivotal role in streaming databases, particularly in the context of reconfiguring stateful operators. One commonly used method is the "stop-and-restart" approach, which is a kind of redo strategy. In this method, program execution is temporarily halted, and the state is securely transferred during this computational pause. Subsequently, the job is restarted once the redistribution of state is complete, often making effective use of existing fault-tolerance mechanisms within the systems [23, 12]. In many reconfiguration scenarios, only a small subset of operators requires state migration. Operators not involved in the migration can continue functioning without disruption, and fault-tolerance checkpoints can be utilized to facilitate the process of state migration [42, 108]. An optimization to this method involves subdividing the state and migrating its partitions [59].

Riveter exhibits orthogonality to live or state migration, augmenting their capabilities within resource-dynamic environments. Riveter facilitates query migration as opposed to full database migration, employing a pipeline-level suspension and resumption strategy. This approach significantly improves the feasibility of live migration in scenarios where resources are ephemeral or fluctuating. Migrating queries prove to be considerably less resource-intensive than relocating entire database states, primarily due to the smaller intermediate states required for serialization and transfer.

4.4.2 Recovery, Checkpoint, and Suspension

Recovery, checkpoint, and suspension can be traced from the same lineage in the realm of databases. In the context of database recovery, a canonical and widely-used algorithm is ARIES [114], which supports steal and no-force buffer approaches, fine-granularity, write-ahead logging, partial rollbacks, and fuzzy checkpoint. Some algorithms proposed to extend the original ARIES algorithm [115]. Although many disk-resident database systems implement ARIES-alike mechanisms for recovery purposes, in-memory database systems avoid using ARIES-style for performance reasons. Commonly, in-memory databases access persistent storage only for recovery thus minimizing the I/O overhead is prioritized in most in-

memory databases [106].

The checkpoint mechanism plays a crucial role in enhancing recovery processes. This mechanism establishes a reference point from which the execution engine can persistently capture the current state of the database. Variations of this checkpoint approach include Transaction Consistent Checkpoint, Action Consistent Checkpoint, and Fuzzy Checkpoint [50]. In contrast to traditional disk-resident database systems, main-memory databases employ an asynchronous approach to transmit transaction updates from the log file to a designated checkpoint archive. This not only serves to expedite recovery but also frees up valuable log space. The checkpoint operation effectively materializes logical operations recorded in the log file into the checkpoint archive [134]. Instead of propagating individual log records, most main-memory databases produce a form of consistent checkpoint often referred to as a snapshot. These snapshots represent the tangible state of the database at a specific moment in time and are generated periodically and asynchronously [132]. The snapshot-based approaches enable in-memory databases [83] and modern disk-resident databases [51] to quickly reload the most recent snapshot in the event of a system crash.

In light of checkpoint mechanisms, Chandramouli et al. introduced a pull-based query execution approach that operates at the operator level and focuses on query suspension and resumption. This approach involves creating a query suspend plan to minimize the overhead associated with suspending and resuming queries. This plan may entail a combination of preserving the current state and reverting to previous checkpoints. PROQID [52] is proposed to suspend and resume queries in distributed environments. A systematic study is conducted to analyze the performance implications of different management policies for long-running query workloads, including policies like ”kill and restart” and ”query and resume.” [78]. Additionally, Graefe et al. and Antonopoulos et al. explored the possibilities of implementing pause and resume functionality for index operations in commercial database systems. In a related context, Amber was introduced to support responsive debugging during the execution of a dataflow [79]. It accomplishes this by transforming an operator DAG (Directed Acyclic Graph) into an actor DAG. The concept of query suspension and resumption is applied within the actor DAG, enabling the execution

to be paused and resumed by exchanging messages such as ”suspend” and ”resume” among the actors involved. Similar to query suspension, the query preemption mechanism can pause or checkpoint some ongoing jobs in favor of others due to specific objectives. SaGe [111] focuses on SPARQL queries and allows the queries to be suspended by the Web server after a fixed period and resumed upon client request. Rotary [96] is a general resource preemption and arbitration framework, but one of its implementations, Rotary-AQP can checkpoint some long-running AQP jobs during execution and resume it using the persisted state when it is beneficial to do so.

Riveter shares a similar idea with the line of recovery, checkpoint and suspension in database systems, yet Riveter is an adaptive framework that can identify optimal solutions to suspend and resume queries. Furthermore, Riveter can check and exploit available resources dynamically when suspending and resuming which maximizes scheduling flexibility.

4.4.3 Query Scheduling

Query scheduling plays a vital role in the architecture of data systems deployed within cloud environments. A prime use case is seen in multi-tenant databases that enable the allocation of limited resources to serve multiple database tenants simultaneously [40]. Within multi-tenant databases, one of the utmost responsibilities is to guarantee that each tenant receives sufficient resources to handle their requests within a specified timeframe, often referred to as a Service Level Objective (SLO). Various approaches exist to address this challenge, such as resource isolation [119], which involves setting aside a fixed or minimal amount of necessary resources for each tenant, as well as intelligent tenant placement [99]. However, as the prevalence of long-running queries grows, managing this responsibility becomes progressively challenging. This is primarily because long-running queries have the potential to exhaust the virtualized resources. For instance, current methods for scheduling long-running queries generally follow one of two strategies: they either attempt to allocate substantial resources to these queries, potentially accelerating the saturation of resources, or they seek to reposition the queries, which can result in increased query latency. Recent research endeavors have explored the possibility of preempting certain long-running jobs

during their execution in favor of prioritizing others [96].

Riveter stands out by acknowledging the dynamic nature of cloud resources, whereas traditional methods assume resource stability. Riveter improves existing solutions significantly by breaking down long-running queries into smaller tasks during suspension and resumption, leading to more efficient query scheduling and resource management.

4.5 Discussion

We discuss implementation choices and open questions for Riveter in this section.

More Suspension and Resumption Strategies. Riveter serves as an adaptable query suspension and resumption framework capable of accommodating a wide range of strategies. In our implementation, we have showcased three distinct strategies: redo, pipeline-level, and process-level, in order to demonstrate Riveter's functionality and performance. Nevertheless, Riveter is versatile and can readily support additional strategies while adapting to more complex scenarios. For example, it is valuable to consider implementing a data-level strategy that can partition input datasets and execute queries in batch mode, particularly when developing a suspension-oriented query execution engine proves to be complicated. Alternatively, an operator-level strategy can be implemented to offer finer-grained suspension capabilities for scenarios involving iterator-based query execution.

Multiple Suspensions during Query Execution. While the proposed Riveter framework can be easily extended to accommodate scenarios with multiple suspension events, our performance evaluation of Riveter mainly focuses on scenarios that involve a single suspension and resumption event. This choice is motivated by the fact that each suspension event can be treated as an independent occurrence, and latency increases proportionally with the number of suspensions. Therefore, it is both reasonable and practical for Riveter to assess the current available latency and potential latency when determining the optimal strategy for handling the current suspension request. As part of our future work, we will also explore more complex scenarios involving multiple queries, where the individual suspension and resumption of queries can potentially interact with one another.

Suspension-friendly Data Layout. In our implementation and evaluation, we operate under the assumption that the data layout is typically stored in its native format. However, there are cases where data layouts can be pre-processed. For example, data can be sorted before execution, and Riveter can leverage this pre-sorted data layout while continuously tracking the watermark during execution. More specifically, the watermark can serve as intermediate data for suspension and resumption, resulting in a significant reduction in the size of intermediate data. Furthermore, this approach provides an opportunity to suspend queries at a finer-grained level, such as at the tuple level.

Query Re-optimization during Suspension. Riveter also raises an intriguing question regarding query re-optimization during suspension. Currently, Riveter always assumes that query plans remain the same when suspending and resuming queries. However, it presents a challenging yet beneficial avenue to explore the possibility of altering the query plan during suspension. This leads to the question of how Riveter should determine the optimal strategy for query suspension and the nature of intermediate data persistence to maximize the benefits of query re-optimization.

CHAPTER 5

CONCLUSION

In modern cloud environments, dynamic and ephemeral resources are rising, exemplified by spot instances and "zero-carbon clouds". Such cloud resources can fluctuate in resource availability and monetary cost, which poses significant challenges: resource scarcity and potential job termination due to resource unavailability or price fluctuations for multi-tenant environments. This dissertation proposes three optimizations for the challenges: resource utilization maximization, resource arbitration, and resource suspension and resumption, and further develops three prototype systems to realize them.

Firstly, to explore resource utilization maximization, we analyze the benefits and limitations of packing multiple models together to take advantage of available GPU resources for model training. Under the proper conditions, this packing can bring up to 40% reduction in latency per model packed, compared with training the models sequentially on a GPU. We further demonstrate that `pack` primitive can be used to accelerate a state-of-the-art hyperparameter tuning algorithm. Our end-to-end tuning system demonstrates a 2.7x speedup in terms of time to find the best model by improving GPU utilization. Our analysis opens many interesting optimization opportunities, such as the training process can be decomposed and scheduled for packing to reduce the overall training time or trading off accuracy or training time to improve overall resource utilization.

Secondly, we argue 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 validate 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.

Finally, we argue that ephemeral resources that fluctuate in availability and monetary cost are increasingly prevalent in modern cloud infrastructure, which necessitates the development of an adaptive query execution mechanism for these scenarios, as exemplified by modern cloud-native databases. We present Riveter, a framework designed for adaptive query suspension and resumption. It offers a spectrum of strategies for suspending queries at varying levels of granularity and subsequently resuming them when deemed necessary or advantageous. To determine the optimal strategies for query suspension, we also design a cost model tailored to the characteristics of these suspension strategies. We conduct a series of evaluations using the TPC-H benchmark, including a performance study of the suspension and resumption strategies implemented in Riveter to present the difference among the strategies in terms of the size of persisted intermediate data; an in-depth analysis of end-to-end pipelines to confirm the adaptive and efficient query suspension and resumption provided by Riveter, and a cost model evaluation showcasing its effectiveness on the estimation accuracy and running time of the cost model.

This dissertation presents resource-aware optimizations to tackle challenges posed by ephemeral resources in multi-tenant environments, and further exploration in this area necessitates a rethinking of the design and development principles of current data-intensive systems to address the dynamic nature of cloud resources and empower data-intensive systems to thrive in these ever-changing environments, ultimately shaping the future of cloud-based data-intensive systems.

REFERENCES

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *IEEE International Conference on Data Engineering (ICDE)*, pages 466–475, 2007.

[2] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael I. Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 481–492, 2014.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[4] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *VLDB Endowment*, 14(12):2986–2998, 2021.

[5] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Godwal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen 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] Amazon EC2 Spot Instances. `https://aws.amazon.com/ec2/spot`. Accessed: 2023-09-10.

[7] Amazon EC2 Spot Instances Pricing. `https://aws.amazon.com/ec2/spot/pricing/`. Accessed: 2023-09-10.

[8] Apache Kafka. `https://kafka.apache.org`. Accessed: 2023-09-29.

[9] Apache Parquet. `https://parquet.incubator.apache.org/`. Accessed: 2023-09-29.

[10] Apache Spark. `https://spark.apache.org`. Accessed: 2023-09-29.

[11] Apache Spark Streaming Checkpointing. `https://spark.apache.org/docs/latest/streaming-programming-guide.html`. Accessed: 2023-07-04.

[12] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *ACM International Conference on Management of Data (SIGMOD)*, pages 601–613, 2018.

[13] Malay Bag, Alekh Jindal, and Hiren Patel. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[14] Sean Kenneth Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Prashant J. Shenoy. ”cut me some slack”: latency-aware live migration for databases. In *International Conference on Extending Database Technology (EDBT)*, pages 432–443, 2012.

[15] Yoshua Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures. In *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700, pages 437–478. Springer, 2012.

[16] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[17] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A Transactional Record Manager for Shared Flash. In *Conference on Innovative Data Systems Research (CIDR)*, pages 9–20, 2011.

[18] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[19] Thomas Bradley. NVIDIA Hyper-Q Example. `http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf`, 2013.

[20] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR*, abs/1605.07678, 2016.

[21] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *IEEE International Conference on Data Engineering (ICDE)*, pages 2859–2872, 2022.

[22] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2477–2489, 2021.

[23] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *VLDB Endowment*, 10(12):1718–1729, 2017.

[24] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. Query suspend and resume. In *ACM International Conference on Management of Data (SIGMOD)*, pages 557–568, 2007.

[25] Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems*, 32(2):9, 2007.

[26] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate Query Processing: No Silver Bullet. In *ACM International Conference on Management of Data (SIGMOD)*, pages 511–519, 2017.

[27] C. L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, 275:314–347, 2014.

[28] Andrew A. Chien. Driving the Cloud to True Zero Carbon. *Communications of the ACM*, 64(2):5, 2021.

[29] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser N. Tantawi, and Chandra Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[30] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[31] Cost-based Optimizer. `https://docs.databricks.com/en/optimizations/cbo.html`. Accessed: 2023-09-29.

[32] CRIU: A project to implement checkpoint/restore functionality for Linux. `https://github.com/checkpoint-restore/criu`. Accessed: 2023-07-04.

[33] Andrew Crotty, Alex Galakatos, Connor Luckett, and Ugur Çetintemel. The Case for In-Memory OLAP on "Wimpy" Nodes. In *IEEE International Conference on Data Engineering (ICDE)*, pages 732–743, 2021.

[34] Carlo Curino, Djellel Eddine Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *ACM Symposium on Cloud Computing (SoCC)*, pages 2:1–2:14, 2014.

[35] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5, 2013.

[36] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *VLDB Endowment*, 4(8):494–505, 2011.

[37] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *UCSB Computer Science Technical Report*, 2010.

[38] DuckDB is an in-process SQL OLAP Database Management System. `https://github.com/duckdb/duckdb`. Accessed: 2023-09-05.

[39] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *VLDB Endowment*, 12(12):2218–2229, 2019.

[40] Aaron J. Elmore, Carlo Curino, Divyakant Agrawal, and Amr El Abbadi. Towards Database Virtualization for Database as a Service. *VLDB Endowment*, 6(11):1194–1195, 2013.

[41] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *ACM International Conference on Management of Data (SIGMOD)*, pages 301–312, 2011.

[42] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM International Conference on Management of Data (SIGMOD)*, pages 725–736, 2013.

[43] Yupeng Fu and Chinmay Soman. Real-time Data Infrastructure at Uber. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2503–2516, 2021.

[44] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R. Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *European Conference on Computer Systems (EuroSys)*, pages 4:1–4:13, 2018.

[45] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1487–1495, 2017.

[46] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. MIT Press, 2016.

[47] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610, 2005.

[48] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 485–500, 2019.

[49] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert M. Patton. FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[50] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, 1983.

[51] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *ACM International Conference on Management of Data (SIGMOD)*, pages 877–892, 2020.

[52] Jon Olav Hauglid and Kjetil Nørvåg. PROQID: partial restarts of queries in distributed databases. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 1251–1260, 2008.

[53] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision (ECCV)*, volume 9908, pages 630–645, 2016.

[56] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *ACM International Conference on Management of Data (SIGMOD)*, pages 171–182, 1997.

[57] Herodotos Herodotou and Elena Kakoulli. Trident: Task Scheduling over Tiered Storage Systems in Big Data Platforms. *VLDB Endowment*, 14(9):1570–1582, 2021.

[58] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[59] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *VLDB Endowment*, 12(9):1002–1015, 2019.

[60] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*, abs/1704.04861, 2017.

[61] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[62] Xin Sunny Huang, Ang Chen, and T. S. Eugene Ng. Green, Yellow, Yield: End-Host Traffic Scheduling for Distributed Deep Learning with TensorLights. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 430–437, 2019.

[63] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR*, abs/1602.07360, 2016.

[64] K. R. Jayaram, Vinod Muthusamy, Parijat Dube, Vatche Ishakian, Chen Wang, Benjamin Herta, Scott Boag, Diana Arroyo, Asser N. Tantawi, Archit Verma, Falk Pollok, and Rania Khalaf. FfDL: A Flexible Multi-tenant Deep Learning Platform. In *ACM/IFIP International Middleware Conference (Middleware)*, pages 82–95, 2019.

[65] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 947–960, 2019.

[66] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Transactions on Architecture and Code Optimization*, 15(3):37:1–37:26, 2018.

[67] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR*, abs/1902.03383, 2019.

[68] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *ACM International Conference on Management of Data (SIGMOD)*, pages 631–646, 2016.

[69] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2232–2245, 2022.

[70] Steven Kay. *Fundamentals of statistical signal processing*. Prentice Hall PTR, 1993.

[71] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C. Mohan. Firestore: The NoSQL Serverless Database for the Application Developer. In *IEEE International Conference on Data Engineering (ICDE)*, pages 3376–3388, 2023.

[72] Jingwoong Kim, Minkyu Kim, Heungseok Park, Ernar Kusdavletov, Dongjun Lee, Adrian Kim, Ji-Hoon Kim, Jung-Woo Ha, and Nako Sung. CHOPT: Automated Hyperparameter Optimization Framework for Cloud-Based Machine Learning Platforms. *CoRR*, abs/1810.03527, 2018.

[73] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *ACM International Conference on Management of Data (SIGMOD)*, pages 691–706, 2015.

[74] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter R. Pietzuch. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *VLDB Endowment*, 12(11):1399–1413, 2019.

[75] Sanjay Krishnan, Aaron J. Elmore, Michael J. Franklin, John Paparrizos, Zechao Shang, Adam Dziedzic, and Rui Liu. Artificial Intelligence in Resource-Constrained and Shared Environments. *ACM SIGOPS Operating Systems Review*, 53(1):1–6, 2019.

[76] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1106–1114, 2012.

[78] Stefan Krompass, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing long-running queries. In *International Conference on Extending Database Technology (EDBT)*, volume 360, pages 132–143, 2009.

[79] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. Amber: A Debuggable Dataflow System Based on the Actor Model. *VLDB Endowment*, 13(5):740–753, 2020.

[80] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *CoRR*, abs/1807.02037, 2018.

[81] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceeding of IEEE*, 86(11):2278–2324, 1998.

[82] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist`. Accessed: 2023-09-29.

[83] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. Index Checkpoints for Instant Recovery in In-Memory Database Systems. *VLDB Endowment*, 15(8):1671–1683, 2022.

[84] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *ACM International Conference on Management of Data (SIGMOD)*, pages 743–754, 2014.

[85] Feifei Li. Cloud native database systems at Alibaba: Opportunities and Challenges. *VLDB Endowment*, 12(12):2263–2272, 2019.

[86] Kaiyu Li and Guoliang Li. Approximate Query Processing: What is New and Where to Go? - A Survey on Approximate Query Processing. *Data Science and Engineering*, 3(4):379–397, 2018.

[87] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[88] Liam Li, Evan R. Sparks, Kevin G. Jamieson, and Ameet Talwalkar. Exploiting Reuse in Pipeline-Aware Hyperparameter Tuning. *CoRR*, abs/1903.05176, 2019.

[89] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18:185:1–185:52, 2017.

[90] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *VLDB Endowment*, 11(5):607–620, 2018.

[91] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline. In *ACM Symposium on Cloud Computing (SoCC)*, pages 61–73, 2019.

[92] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR*, abs/1807.05118, 2018.

[93] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. *arXiv preprint arXiv:1802.07389*, 2018.

[94] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. In *Conference on Machine Learning and Systems (MLSys)*, 2019.

[95] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron J. Elmore, and Shan-Hung Wu. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems. *VLDB Endowment*, 12(5):597–610, 2019.

[96] Rui Liu, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. Rotary: A Resource Arbitration Framework for Progressive Iterative Analytics. In *IEEE International Conference on Data Engineering (ICDE)*, pages 2140–2153, 2023.

[97] Rui Liu, Sanjay Krishnan, Aaron J. Elmore, and Michael 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)*, pages 3:1–3:11, 2021.

[98] Rui Liu, David Wong, Dave Lange, Patrik Larsson, Vinay Jethava, and Qing Zheng. Accelerating container-based deep learning hyperparameter optimization workloads. In *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, pages 6:1–6:10, 2022.

[99] Ziyang Liu, Hakan Hacigümüs, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. PMAX: tenant placement in multitenant databases for profit maximization. In *International Conference on Extending Database Technology (EDBT)*, pages 442–453, 2013.

[100] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. From Batch Processing to Real Time Analytics: Running Presto® at Scale. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1598–1609, 2022.

[101] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *VLDB Endowment*, 15(11):3098–3111, 2022.

[102] Minghuang Ma, Hadi Pouransari, Daniel Chao, Saurabh Adya, Santiago Akle Serrano, Yi Qin, Dan Gimnicher, and Dominic Walsh. Democratizing Production-Scale Distributed Deep Learning. *CoRR*, abs/1811.00143, 2018.

[103] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *European Conference on Computer Vision (ECCV)*, volume 11218, pages 122–138, 2018.

[104] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2177–2190, 2022.

[105] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 142–150, 2011.

[106] Arlino Magalhães, José Maria Monteiro, and Angelo Brayner. Main Memory Database Recovery: A Survey. *ACM Computing Survey*, 54(2):46:1–46:36, 2022.

[107] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling for Machine Learning Workloads. *CoRR*, abs/1907.01484, 2019.

[108] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *VLDB Endowment*, 11(10):1303–1316, 2018.

[109] Dominic Masters and Carlo Luschi. Revisiting Small Batch Training for Deep Neural Networks. *CoRR*, abs/1804.07612, 2018.

[110] Leonel Aguilar Melgar, David Dao, Shaoduo Gan, Nezihe M Gürel, Nora Hollenstein, Jiawei Jiang, Bojan Karlaš, Thomas Lemmin, Tian Li, Yang Li, Susie Rao, Johannes Rausch, Cedric Renggli, Luka Rimanic, Maurice Weber, Shuai Zhang, Zhikuan Zhao, Kevin Schawinski, Wentao Wu, and Ce Zhang. Ease.ML: A Lifecycle Management System for MLDev and MLOps. In *Conference on Innovative Data Systems Research (CIDR)*, 2021.

[111] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. SaGe: Web Preemption for Public SPARQL Query Services. In *The World Wide Web Conference (WWW)*, pages 1268–1278, 2019.

[112] Takeshi Mishima and Yasuhiro Fujiwara. Madeus: Database Live Migration Middleware under Heavy Workloads for Cloud Environment. In *ACM International Conference on Management of Data (SIGMOD)*, pages 315–329, 2015.

[113] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[114] C. Mohan and Frank E. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *ACM International Conference on Management of Data (SIGMOD)*, pages 371–380, 1992.

[115] C. Mohan and Inderpal Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *ACM International Conference on Management of Data (SIGMOD)*, pages 55–66, 1994.

[116] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[117] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *VLDB Endowment*, 13(11):2159–2173, 2020.

[118] Vivek R. Narasayya and Surajit Chaudhuri. Multi-Tenant Cloud Data Services: State-of-the-Art, Challenges and Opportunities. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2465–2473, 2022.

[119] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[120] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 2019.

[121] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning*, 2018.

[122] Thomas Neumann and Michael J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[123] NVIDIA System Management Interface. `https://developer.nvidia.com/nvidia-system-management-interface`. Accessed: 2023-09-29.

[124] NVIDIA Virtual GPU. `https://www.nvidia.com/en-us/data-center/virtual-solutions`. Accessed: 2023-09-29.

[125] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *European Conference on Computer Systems (EuroSys)*, pages 3:1–3:14, 2018.

[126] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A Scalable Query Engine on Cloud Functions. In *ACM International Conference on Management of Data (SIGMOD)*, pages 131–141, 2020.

[127] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. *VLDB Endowment*, 15(6):1279–1287, 2022.

[128] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric N. Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. Cloud-Native Transactions and Analytics in SingleStore. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2340–2352, 2022.

[129] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[130] Mark Raasveldt and Hannes Mühleisen. DuckDB: an Embeddable Analytical Database. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1981–1984, 2019.

[131] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, page 7, 2012.

[132] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1539–1551, 2016.

[133] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[134] Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. In *IEEE International Conference on Data Engineering (ICDE)*, pages 452–462, 1989.

[135] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[136] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. ProRea: live database migration for multi-tenant RDBMS with snapshot isolation. In *International Conference on Extending Database Technology (EDBT)*, pages 53–64, 2013.

[137] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *CoRR*, abs/1907.10597, 2019.

[138] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *CoRR*, abs/1804.10001, 2018.

[139] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1802–1813, 2019.

[140] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Survey*, 54(11s):239:1–239:32, 2022.

[141] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. CrocodileDB: Efficient Database Execution through Intelligent Deferment. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[142] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[143] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *ACM Symposium on Cloud Computing (SoCC)*, pages 368–380, 2015.

[144] Logan Stafman, Andrew Or, and Michael J. Freedman. ReLAQS: Reducing Latency for Multi-Tenant Approximate Queries via Scheduling. In *ACM/IFIP International Middleware Conference (Middleware)*, pages 280–292, 2019.

[145] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3645–3650, 2019.

[146] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Modern Deep Learning Research. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 13693–13696, 2020.

[147] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[148] Seyed Ali Mohammad Tajalli, Seyede Zahra Tajalli, Maryam Homayounzadeh, and Mohammad Hassan Khooban. Zero-Carbon Power-to-Hydrogen Integrated Residential System Over a Hybrid Cloud Framework. *IEEE Transactions on Cloud Computing*, 11(3):3099–3110, 2023.

[149] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning (ICML)*, volume 97, pages 6105–6114, 2019.

[150] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Thrifty Query Execution via Incrementability. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1241–1256, 2020.

[151] TensorFlow. `https://www.tensorflow.org`. Accessed: 2023-09-29.

[152] TensorFlow Basics. `https://www.tensorflow.org/tutorials/customization/basics`. Accessed: 2023-09-29.

[153] TPC-DS Benchmark. `https://www.tpc.org/tpcds/default5.asp`. Accessed: 2023-09-11.

[154] TPC-H Benchmark. `https://www.tpc.org/tpch/default5.asp`. Accessed: 2023-09-11.

[155] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[156] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *European Conference on Computer Systems (EuroSys)*, pages 35:1–35:16, 2016.

[157] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR*, abs/1908.08962, 2019.

[158] Universal dependencies. `https://universaldependencies.org`. Accessed: 2023-09-29.

[159] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing (SOCC)*, pages 5:1–5:16, 2013.

[160] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[161] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-Tuning Query Scheduling for Analytical Workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1879–1891, 2021.

[162] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[163] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, Chengjun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proceedings of the ACM on Management of Data*, 1(2):199:1–199:25, 2023.

[164] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 41–53, 2018.

[165] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. MotherNets: Rapid Deep Ensemble Learning. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[166] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 335–347, 2017.

[167] Eugene Wu, Samuel Madden, Yang Zhang, Evan Jones, and Carlo Curino. Relational cloud: The case for a database service. *MIT CSAIL Technical Report*, 2010.

[168] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, 2018.

[169] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–548, 2020.

[170] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.

[171] Hidehito Yabuuchi, Daisuke Taniwaki, and Shingo Omura. Low-latency Job Scheduling with Preemption for the Development of Deep Learning. In *USENIX Conference on Operational Machine Learning (OpML)*, pages 27–30, 2019.

[172] Peifeng Yu and Mosharaf Chowdhury. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[173] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision (ECCV)*, volume 8689, pages 818–833, 2014.

[174] Kai Zeng, Sameer Agarwal, and Ion Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1347–1361, 2016.

[175] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 1049–1062, 2019.

[176] Junzhe Zhang, Sai-Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient Memory Management for GPU-based Deep Learning Systems. *CoRR*, abs/1903.06631, 2019.

[177] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. CompuCache: Remote Computable Caching using Spot VMs. In *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[178] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6848–6856, 2018.

[179] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *USENIX Symposium on Operating Systems Design and Implementation OSDI*, pages 515–532, 2020.

[180] Haoyue Zheng, Fei Xu, Li Chen, Zhi Zhou, and Fangming Liu. Cynthia: Cost-Efficient Cloud Resource Provisioning for Predictable Distributed Deep Neural Network Training. In *International Conference on Parallel Processing (ICPP)*, pages 86:1–86:11, 2019.