UNIVERSITY OF CALIFORNIA SAN DIEGO

High-throughput Data Systems for Deep Learning Workloads

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Yuhao Zhang

2023

University of California San Diego

2023

DEDICATION

To Xiaowen,
my love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my PhD advisor, Prof. Arun Kumar, for his guidance, support, and mentorship throughout the entirety of my study at UCSD, especially the darker days of the 2019-2022 global pandemic. I owe my career to him after switching tracks from physics to computer science. Prof. Kumar has been an invaluable source of inspiration, pushing me to explore new frontiers in my research and providing invaluable insights that have significantly enriched the quality of my work. His commitment to excellence contributes greatly to this dissertation, and I am indebted to him for his profound impact on my academic and personal growth.

I want also to extend my heartfelt gratitude to the members of my PhD committee, Prof. Alin Deutsch, Prof. Yusu Wang, and Prof. Yiying Zhang. Their expertise and constructive feedback have been instrumental in shaping the direction of my dissertation. I sincerely appreciate the time and effort they dedicated to reviewing my work, offering insightful advice, and challenging me to think critically.

Further, I want to express my sincere appreciation to my co-authors, Supun Nakandala, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, Side Li, Advitya Gemawat, and Kabir Nagrecha, for their invaluable collaboration and intellectual contributions through our collaborative research work, and I am grateful to have a chance to work with such talented and committed individuals.

I am appreciative of my internship mentors, Qi Zhao, Yan Rui, Rathijit Sen, and Konstantinos Karanasos. Their guidance, expertise, encouragement, and willingness to share their knowledge have been influential in my professional growth. I am grateful for the opportunities they provided for hands-on learning to hone my skills and the constructive feedback and insightful advice that extended beyond the workplace.

I am fortunate to be part of Arun's lab and the UCSD database group. It was a refreshing experience to be in such a cohort of talented and motivated researchers and students. I am thankful for the shared challenges, the exchange of ideas, and the diverse perspectives each person brings. Beyond the academic realm, the friendships formed are invaluable, making this pursuit not only intellectually stimulating but also personally fulfilling. I want to say thank you to Supun Nakandala, Vraj Shah, Side Li, Kabir Nagrecha, Kyle Luoma, Xiuwen Zheng, Tara Mirmira, Rana Alotaibi, Liangde Li, Advitya Gemawat, Vignesh Nanda Kumar, and Pradyumna Sridhara.

Finally, I want to thank my family for their support and unconditional love throughout the years. This achievement is as much theirs as it is mine.

This dissertation is based on and contains materials from the following publications.

Chapter 3 contains material from “Cerebro: A Data System for Optimized Deep Learning Model Selection” by Supun Nakandala, Yuhao Zhang, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 13, Issue 12, July 2020. The dissertation author’s contribution was in the conceptualization of the system, parts of the implementation, and parts of the experiments.

Chapter 4 contains material from “Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches” by Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 14, Issue 10, July 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 5 contains material from “Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines” by Yuhao Zhang and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 16, Issue 11, August 2023. The dissertation author was the primary investigator and author of this paper.

Chapter 6 contains material from “Panorama: A Data System for Unbounded Vocabulary Querying over Video” by Yuhao Zhang and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 13, Issue 4, December 2019. The dissertation author was the primary investigator and author of this paper.

My co-authors have kindly approved the inclusion of the aforementioned publications in my dissertation.

VITA

2016	BS in Theoretical Physics, Nankai University, China
2021	MS in Computer Science, University of California San Diego
2023	PHD in Computer Science, University of California San Diego

ABSTRACT OF THE DISSERTATION

High-throughput Data Systems for Deep Learning Workloads

by

Yuhao Zhang

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Arun Kumar, Chair

Artificial Intelligence (AI) and Deep Learning (DL) have gained enormous popularity and have seen wide adoption across different domains. They ushered in an era of huge workloads that are increasingly computation- and data-intensive and put existing data analytics infrastructures and systems to the test. However, many of these workloads run with severe inefficiency and face tremendous scalability challenges due to suboptimal scheduling and poor resource/memory management, resulting in wasted computational and storage resources. Furthermore, DL workloads have been predominantly run on custom software frameworks far away from where most enterprise and operational data resides – databases and data systems. We realize that a significant gap exists between existing data systems and DL workloads. Data is often stored in the former but needs to be frequently exported. These large data movements between data and DL systems waste storage, network, and time. It also creates difficulties in data governance, provenance tracking, and compliance with data privacy regulations. Most importantly, large-scale data systems used to be at the center of data analytics before the recent takeover by DL, but many of the lessons and techniques would still apply to DL workloads, and there are missed opportunities to innovate upon existing infrastructures. This dissertation will focus on modern DL workloads and these system efficiency, scalability, and practicality challenges. We aim to raise the throughput of DL systems at a large data scale without sacrificing practicality using a central methodology of *reimagining DL systems as DL data systems*. On the one hand, we apply and innovate techniques inspired by database management systems, such as multi-query optimizations, query plan rewrites, and approximate processing, for DL workloads. On the other hand, we explore novel ways to extend existing data systems, without modifications to their core codebase, to run DL workloads, both bridging the gap and offering tangible data scalability benefits. All proposed techniques and systems are empirically tested and demonstrated to show improvements, sometimes over 10x, compared to state-of-the-art solutions.

Chapter 1

Introduction

1.1 Motivation and Goals

Artificial Intelligence (AI) has ushered in a new era of innovation and revolution in subjects ranging from machine translation [276], face recognition [254], natural sciences [101], to even eerily human-like chatbots [54]. Deep Learning (DL), the backbone of modern AI, has gained tremendous attention and has seen wide adoption from academia to the industry. Advancements in DL have created workloads that put existing data analytics infrastructures and systems to the test. The huge DL workloads are growing increasingly computation- and data-intensive. Among all the systems challenges created by modern DL, this dissertation will focus on three of the most acute issues: speed, scalability, and practicality. We formulate the overall goal of this dissertation work as *to raise throughput of DL systems at a large data scale without sacrificing practicality*. We will now explain the motivation and goals in detail.

The need for speed. Despite their rapid adoption, many DL workloads run with severe inefficiency and face huge scalability issues due to suboptimal scheduling, poor resource/memory management, or the lack of proper software system support. Today, accelerators such as GPUs are crucial assets, with state-of-the-art chips and machines growing ever more expensive and power-hungry. Furthermore, with the stagnated Moore's law, large-scale distributed computations are becoming almost inevitable, making all the systems issues even more complex and the cost prohibitively large. Runtime inefficiency and computational resource limitations are among the most significant challenges for both scientific innovations and industrial adoption. Therefore, it is imperative for systems researchers to investigate bottlenecks, eliminate overheads, and accelerate these workloads end-to-end. In this dissertation, we choose throughput as the central metric for speed. Whether it is the number of models per day for model selection workloads, epochs per hour for training, or queries per second for inference, higher throughput leads to substantial cost savings, higher productivity, and better models resulting from faster model selection and training.

The data challenges. Being data-intensive is another defining characteristic of DL. The scale and the amount of data can be tremendous for DL workloads [78, 73, 54]. Data management for modern DL applications is complicated by the great variety of data modalities, ranging from tabular data to videos to even graph data. Each modality comes with its own set of challenges; for example, videos are called “fast” data [142], as they arrive at high speed and demand real-time processing, but they possess high redundancy between frames that can be exploited. On the other hand, graph data has dramatically different characteristics from other data modalities, and the models used on them are wildly different in data access patterns [150, 113]. Overall, systems designed for DL workloads must be able to handle large volumes, TBs or even PBs of, data and communications efficiently. Hence, any DL systems need to be, first and foremost, scalable data systems that can schedule, optimize, and execute large workloads with huge amounts of dataflows.

The real-world constraints. The data challenges mentioned above require data systems solutions. However, a major gap exists between the present data systems, represented by large-scale database management systems (DBMS) and data lakehouses such as Spark, and the DL workloads. Modern DL workloads are more often executed on custom systems with little data layer; they focus on fast hardware kernels and compiler-level optimizations but have little consideration for the logical plans and the practical limitations of data storage, movement, and management. At the same time, data systems have largely treated DL workloads as an afterthought, relegating them almost entirely to external DL systems. There has been a general lack of possibility to run DL workloads efficiently closer to databases, where most business and enterprise data typically resides. This gap is preventing DL from further adoption and creates avoidable heavy data movement between custom DL systems and existing data systems, creating overheads and hindering usability, degree of automation, ease of governance, and data privacy requirements posed by legislations like GDPR [72] and CCPA [221]. Towards this end, the third goal of this dissertation is to ensure the practicality of DL systems and seek opportunities to ease some of the issues by adopting existing data systems and their techniques.

This dissertation identifies the three most important DL workloads to optimize. We briefly introduce them and will review the details in Chapter 2:

1. Model selection. DL models require extensive tuning of hyperparameters and architectures to achieve maximum accuracy and runtimes. These are critical knobs that affect the model's performance substantially. However, their effect is highly non-linear and typically unpredictable, Thus, model selection has primarily remained an empirical process based on trial and error of training dozens to hundreds of models. Given the weeks or even months of development and training time, model selection is among the most significant bottlenecks of DL.

2. Training. DL training predominantly uses the family of mini-batch Stochastic Gradient Descent (SGD) algorithms to minimize training losses on large-scale datasets. As the scale of DL workloads keeps growing, computational costs have skyrocketed, and parallel and distributed training has become the norm. Any optimizations in the training process can contribute to overall considerable savings in computation and allow for faster model development and selection.

3. Inference. When tuned and trained, the model will be deployed and used to answer queries on data not seen during training. This process is known as model inference. Inference has much in common with training: they typically share the same data pipelines and the forward pass portion of mini-batch SGD. However, some model inference workloads like

video monitoring can demand high runtime speed and real-time processing. Meanwhile, inference workloads typically have more leeway regarding approximate processing, opening up speed-up opportunities. Therefore, the trade-offs between accuracy and throughput are a constant theme in model inference systems.

This dissertation will focus on three representative data modalities and their applications where DL has seen a lot of success.

1. Unstructured data (e.g., videos and images). DL has achieved tremendous success on unstructured data and enabled many novel applications [142, 78, 237].

2. Tabular data. The default form of data in most database systems. Most enterprises and business intelligence (BI) tasks rely primarily on tabular data.

3. Graph data. Tabular data and unstructured data, including tables, time-series, videos, and images, can also be roughly categorized as Euclidean and IID data, which have regular shapes (table dimensions, frame/image's rectangular shape sizes and channels, etc.) and are drawn independently from a large population. Graph data, however, is both non-Euclidean [52] and non-IID as the shape (topology) is highly irregular compared to images, and the data within a graph are explicitly connected via edges, nullifying the IID assumption. This brings profound consequences as the regular DL models and their training methods suddenly stop working, and modifications are required. Coupled with the complexity of large-scale graph data management and their huge memory consumption, DL on graph data poses severe challenges to existing systems.

1.2 Technical Contributions

To tackle the various challenges raised by these workloads and data modalities, our central methodology is **reimagining DL systems as DL data systems**. On the one hand, DL systems, at the core, are data systems that coordinate large dataflows and, therefore, can benefit

Table 1.1. Common technical themes used in this dissertation.

	Multi-query Optimizations	Plan Rewrites	Bulk Synchronous Parallelism	Using Existing Data Systems	Similarity Search
Cerebro	✓		✓		
Cerebro on DS	✓		✓	✓	
Lotan	✓	✓	✓	✓	
Panorama		✓			✓

from data systems optimizations. We apply and innovate upon a plethora of DBMS-inspired techniques such as multi-query optimizations [256], multimedia and similarity-based databases and approximate processing [36, 22, 141], and query plan rewriting and optimizations to DL systems, boosting throughput and resource-efficiency. On the other hand, to mitigate the typical lack of data layer in the popular DL tools and to increase the practicality of DL, we also work to bridge the gap between existing data systems and DL workloads, bringing model selection, training, and inference capabilities to existing relational and graph DBMSes.

This dissertation will present in detail our efforts for high-throughput, large-scale, and practical DL data systems and demonstrate the substantial boosts in runtime performance (over 10x in some cases) and scalability (managing workloads that would otherwise fail due to systems crashes, insufficient memory, or exceeding runtime limits) resulted from these technical innovations. Our work spans various DL workloads and data modalities. Figure 1.1 summarizes the scope and positioning for each project. We summarize the common technical themes in Table 1.1. We will now briefly introduce each project included in this dissertation.

1.2.1 CEREBRO: Multi-query Optimization for High-throughput DL Model Selection

CEREBRO is a system to tackle the throughput problems of model selection. We notice most DL systems focus on training one model at a time, reducing throughput and raising overall resource costs; some also sacrifice reproducibility. Critical optimization opportunities are missed as the training workloads share extensively in data and computation, which resembles the

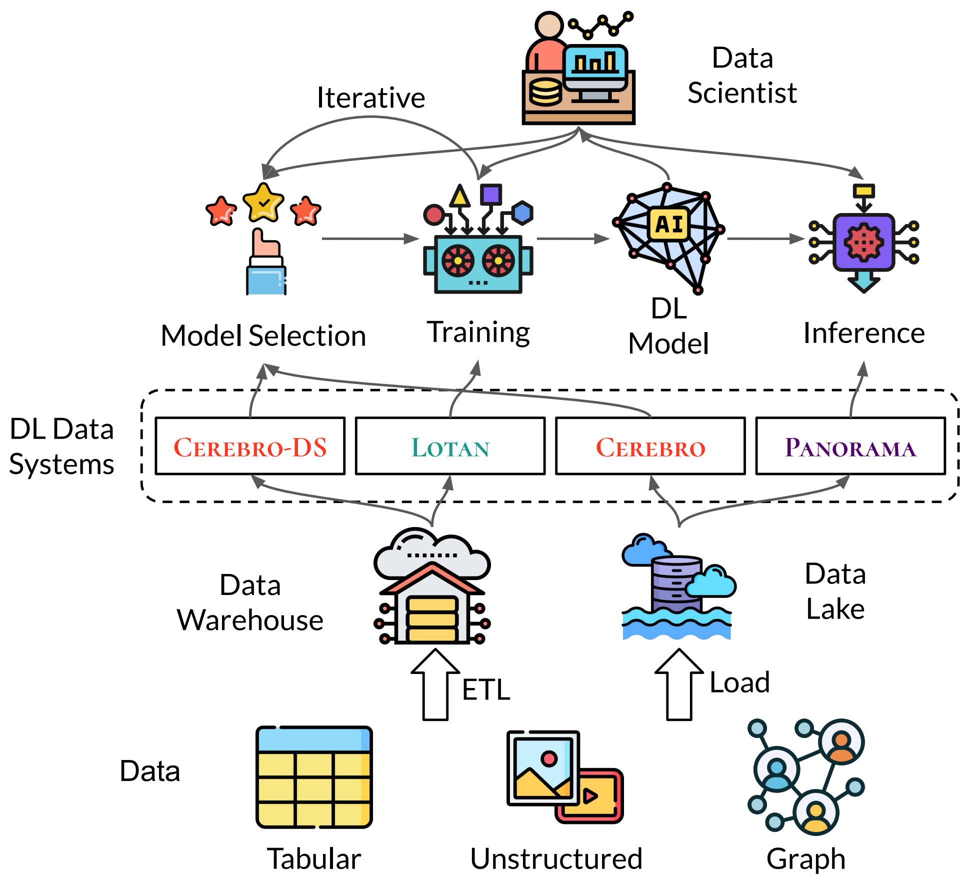


Figure 1.1. The common life cycle of AI/DL applications and an overview of the work covered in this thesis.

multi-query optimization problem in databases research [256]. Towards higher throughput and resource utilization and as a part of a grander vision [161] of DL model selection systems, we built Cerebro [214]. Cerebro is a data system that raises DL model selection throughput at scale, without raising resource costs or sacrificing reproducibility or accuracy. CEREBRO uses a novel parallel DL training strategy called model hopper parallelism. It hybridizes task and data parallelism to mitigate the cons of these prior paradigms and offers the best of both worlds. Experiments on large benchmark datasets showed that Cerebro provides 3x to 10x runtime savings relative to data-parallel systems like Horovod and Parameter Server and up to 8x memory/storage savings or 100x network savings relative to task-parallel systems. Cerebro also supports heterogeneous resources and fault tolerance.

This work will be covered in Chapter 3 and is done jointly with Supun Nakandala and Arun Kumar. A paper on this work has appeared at VLDB 2020 [212]. The system is open source and available at https://github.com/ADALabUCSD/cerebro-system.

1.2.2 CEREBRO on Data Systems: Bridging the Gap between Data Systems and DL Workloads

DL's popularity is not limited to DL researchers; many enterprises and businesses are also considering adopting DL for their data analytics applications. Large business-critical datasets in such settings typically reside in DBMSs or other parallel data systems. In the work of CEREBRO above, we explored the landscape of standalone DL model selection systems and proposed a new parallelism strategy. However, it was unclear if the proposed parallelism and task scheduler could be incorporated into existing infrastructures of data management systems. In this project, we characterized the particular suitability of CEREBRO on data systems. To bring the novel model hopper parallelism approach to DB resident data, we showed that there was no single "best" approach and an interesting tradeoff space exists. We explained four canonical approaches and built prototypes upon the Greenplum Database. We compared them analytically on multiple criteria (e.g., runtime efficiency and ease of governance) with real large-scale DL workloads. The experiments and analyses showed that it was non-trivial to meet all practical desiderata, and there was a Pareto frontier; for instance, some approaches are 3x-6x faster but fare worse on governance and portability. These results and insights can help DBMS and cloud vendors design better DL support for DB users.

Chapter 4 is dedicated to this work. It was done partially during an internship at VMware and jointly conducted with Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. A paper on this work has appeared at VLDB 2021 [319]. The system is open source and available at https://github.com/makemebitter/cerebro-ds. It has also been incorporated into Apache MADlib open source project https://madlib.apache.org/.

1.2.3 LOTAN: Bridging the Gap between Graph Data Systems and Graph Neural Network Workloads

Moving on from the common IID data and corresponding DL workloads that CEREBRO is designed for, we look at the rapidly growing field of Graph Neural Networks (GNN). The complexity of GNN training and various scalability challenges have sparked interest from the machine learning systems community, with efforts to build systems that provide higher efficiency, better memory management, and schemes to reduce costs. However, many of these systems reinvent the wheel by rediscovering years of research and development on advanced graph-parallel data systems. Further, they often couple the scalability challenges of graph data processing with those of GNN training, resulting in entangled complex problems and systems that often fall short on one side of the scalability challenges.

This work proposes LOTAN, a novel and highly scalable data system for full-batch GNN training with a clean decoupling of graph and neural network at its core. With this decoupling, LOTAN can achieve high scalability and bridge the gap between existing graph data systems and DL workloads. LOTAN offers a series of technical innovations inspired by data systems techniques, including execution plan rewriting, highly efficient data movement between systems, a GNN-centric graph partitioning method and the corresponding NN gradient backpropagation scheme, and GNN model batching. Using real large-scale GNN workloads, we demonstrated the system's capability to train large GNN models that prior art, even from industry labs, crashed on. The system can surpass the training throughput of state-of-art systems by up to 40x and beat a naively implemented in-data-system GNN training framework by 76x. Lotan can increase efficiency for existing workloads and open new design freedom for future GNN algorithmic research.

This work will be covered in Chapter 5, and it is jointly conducted with Arun Kumar. A paper on this work has appeared at VLDB 2023 [318]. The system is open source and available at https://github.com/makemebitter/lotan.

1.2.4 PANORAMA: Multimedia DB-style Retrieval with DL Inference

We now take a look at DL inference issues on unstructured data such as videos and images, which have particularly high requirements for real-time processing and bring unique challenges. The prior art has studied how to improve system efficiency. Nevertheless, they primarily focus on small "closed world" prediction vocabularies, even though many surveillance security and traffic analytics applications have an ever-growing set of target entities. We call this the "unbounded vocabulary" issue, which is a crucial bottleneck for emerging video monitoring applications. We presented the first data system for tackling this problem for video querying, PANORAMA. The design philosophy is to build a unified and domain-agnostic system that lets application users generalize to unbounded vocabularies in an out-of-the-box manner without tedious manual re-training. To this end, we synthesized and innovated upon an array of techniques from the literature of ML, vision, databases, and multimedia systems to devise a new system architecture. We also presented designs to ensure PANORAMA had high inference throughput. Experiments with multiple real-world datasets showed that the system could achieve between 2x to 20x higher throughput than baseline approaches on in-vocabulary queries while still yielding comparable accuracy and also generalizing well to unbounded vocabulary queries.

This work is the subject of Chapter 6 and is a joint work with Arun Kumar. A paper on this work appeared at VLDB 2020 [268]. The system is open source and available at https://github.com/makemebitter/Panorama-UCSD.

1.3 Research Impact

Practicality is one of the major foci of this dissertation, and we have put a lot of effort into bringing these research ideas to production with real-world applications. We have had collaborations and conversations with domain scientists, data scientists, ML researchers, and DBMS vendors, and we have helped them adopt our techniques and ideas. These conversations often inspired our work in tackling critical bottlenecks, offering solutions, and providing faster and better DL systems to run these workloads. At the time this document is composed, our research has achieved the following avenues of practical impact:

- CEREBRO has been used by UCSD Public Health researchers for model building. It led to 4 public health journal papers by our collaborators based on models trained with Cerebro, which are now state of the art in their field [5].
- CEREBRO has been reviewed by Databricks and pointed to their customers.
- CEREBRO on Data Systems has been adopted by VMware and shipped in their Greenplum database.
- CEREBRO on Data Systems has been incorporated into Apache MADlib project [196], an in-DBMS ML library.
- A major graph DBMS vendor has expressed interest in collaboration on LOTAN.

Chapter 2

Background

2.1 Deep Learning

The backbone of today's AI marvel is Deep Learning (DL). DL is a subfield of ML inspired by the human brain's structure and function. It involves training artificial neural networks (NN) to perform various tasks ranging from image recognition to natural language processing and many other complex pattern recognition problems. At the core of DL, deep neural networks are composed of layers of interconnected nodes, or "neurons", which have learnable weights that are set during training. Figure 2.1 shows a conceptual drawing of neural networks; in its broadest definition, an NN can take arbitrary inputs, approximate any function on the input, and output arbitrary format of outputs. The use of multiple (even hundreds of) layers and billions of trainable model parameters [117, 172, 54] have led to enormous success in modeling complex and hierarchical relationships within data.

DL models are particularly well-suited for large and high-dimensional datasets but often require a huge amount of training data. Different types of layers have different neuron connections and suitabilities for various tasks due to the inductive bias they implicitly introduce in featurization. Consequently, different model architectures tend to have specialties in data modalities; CNNs are predominantly used in images/videos and vision tasks, LSTMs are particularly suitable for time-series data, and Transformers have an edge on text. As the frontier of DL research rapidly advances, the workloads have grown larger and larger, posing challenges to

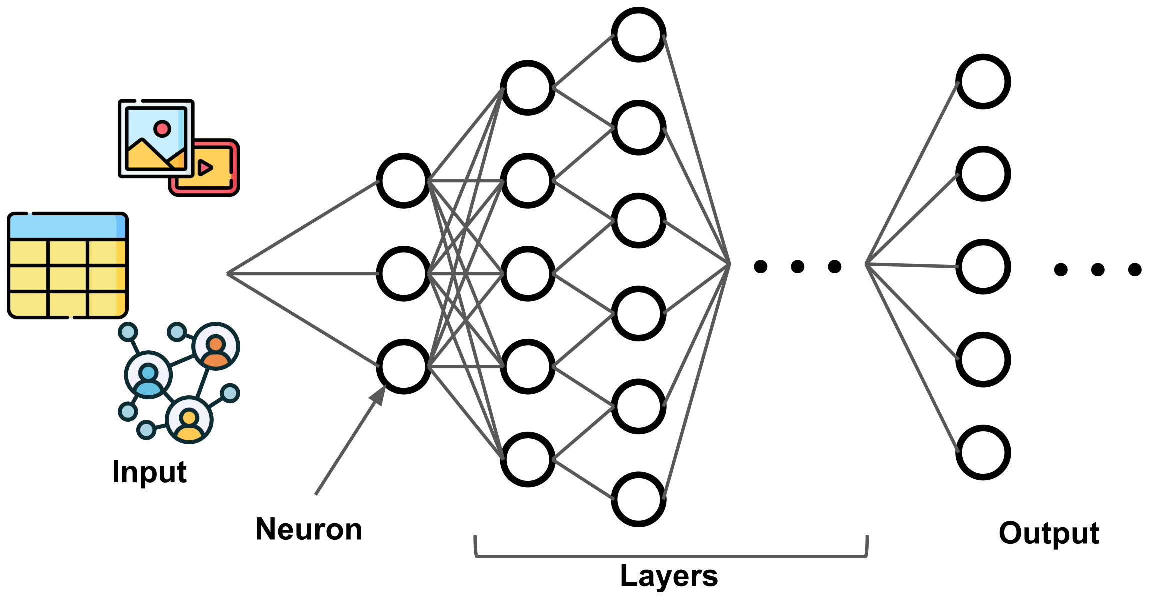


Figure 2.1. A conceptual drawing of a neural network.

the underlying data infrastructure and systems. For example, recent Transformer-based large models [82, 54] have become more computationally intensive than ever, require substantial data, and have since sparked wide interest from the systems community.

2.2 Model Training: Mini-batch Stochastic Gradient Descent

DL training, essentially a non-convex optimization problem [108], is done via algorithms from the Stochastic Gradient Descent (SGD) family for the most part. SGD is an iterative process that performs multiple passes over the data, updating the model parameters along the way. Mini-batch SGD algorithms [149, 39] is a variant of SGD that uses smaller, random subsets of the training data, known as mini-batches, to make the updates. The formula is as follows:

$$\Theta_{new} = \Theta_{old} - \eta \times \nabla L(\Theta_{old}, mini\_batch), \tag{2.1}$$

where Θ_{new} and Θ_{old} are the updated and current model parameters, respectively. η is the learning rate, a hyperparameter that controls the learning step size in the parameter space. $\nabla L(\Theta_{old}, mini-$

batch) is the gradient of the loss function with respect to current model parameters, calculated based on the mini-batch.

It is worth noting that the above largely only applies to IID data, which is the assumption of the sampling-based mini-batch method. However, certain data, such as graphs, are explicitly non-IID. It has profound implications for the training algorithm, the parallelization, and the execution of graph neural network training. Consequently, training a DL model on graph data takes distinctive approaches and requires a very different set of techniques. We will discuss them in more detail in Chapter 5.

2.3 Model Selection

DL is especially complicated because of the vast amount of empirical settings and hyperparameters that are difficult to learn with SGD and usually need to be set by humans. Without a known robust theory to precisely predict a model's accuracy and guide the process, a lot of the effects of the knobs and settings are difficult to gauge beforehand, and many of them have highly non-linear effects. The art of choosing the exemplary model architecture and tuning the hyperparameters is known as model selection. This process is iterative and heavily based on trial and error. Data scientists typically need to try out the combinations of different configurations and train one model for each configuration [258, 160]. In practice, hundreds of models are not unheard of for model selection and are very time-consuming because each model can take hours, if not days [86]. Therefore, end-to-end DL training is about more than just getting one model trained as fast as possible; it is about the throughput of model selection.

2.4 Model Inference

Model inference refers to the process of using trained neural networks to make predictions on new, unseen data. During inference, the model takes input data and produces predictions without updating its model weights. Consequently, inference has much in common with training:

they typically share the same data pipeline and the forward pass portion of mini-batch SGD. Despite the similarities, several unique challenges and considerations arise in this process. There is a stringent demand for high runtime speed in real-world applications, especially those requiring real-time responses, such as online services, video monitoring, and autonomous driving. On the other hand, in the deployed environment, hardware may become the limiting factor as the edge or IoT devices usually lack the computational power to run large and complex models. However, with challenges also come opportunities; inference workloads typically have more leeway regarding approximate processing. Therefore, the trade-offs between accuracy and throughput become a constant theme. Last, there may be regulatory requirements for the deployed models, especially in sensitive domains like healthcare or finance, when customer data is involved. Adhering to privacy laws and data legislation is crucial for model inference systems.

2.5 DL on Database-resident Data

DL's popularity leads to a growing demand for products that make it easier to adopt DL, especially among enterprises [140]. However, large business-critical datasets typically reside in DBMSs or other data systems and are not stored as loose files in data lakes, as most custom DL frameworks assume. One may wonder if DL is useful for DBMS users, since DL is primarily popular on unstructured data, while DBMSs mainly handle tabular data [157]. DBMSs have long provided storage support for text, multimedia [226, 307], and other objects [33, 273]. Furthermore, due to the benefits of embedding learning and less feature engineering in DL [292, 283], many recent works in both research and enterprise applications show that DL is becoming increasingly usable and effective even on structured data [181, 111, 284, 286, 192]. Multimodal analytics combining structured and unstructured data are also popular and relevant for DB users [26, 208, 189, 289]. Finally, DL's "interpretability" pain, once a showstopper for some enterprise users, is being actively mitigated by ML researchers [314, 63]. In the reality of ML practice, data scientists do not think in an all-or-nothing manner; different model types, including

DL, are popular for different use cases. DL is an area that needs more attention from the database community and will be discussed in detail in Chapter 4 and 5.

Chapter 3

CEREBRO: Multi-query Optimization for High-throughput DL Model Selection at Scale

3.1 Introduction

In this chapter, we will go over the many pains and challenges embedded in DL model selection workloads, and propose our system CEREBRO to mitigate them. Our motivation for this work came from our conversations and collaborations with domain researchers, and they have utilized our system ever since.

Case Study. We present a real-world model selection scenario. Our public health collaborators at UC San Diego wanted to try deep nets for identifying different activities (e.g., sitting, standing, stepping, etc.) of subjects from body-worn accelerometer data. The data was collected from a cohort of about 600 people and is labeled. Its size is 864 GB. During model selection, we tried different deep net architectures such as convolution neural networks (CNNs), long short-term memory models (LSTMs), and composite models such as CNN-LSTMs, which now offer state-of-the-art results for multivariate time-series classification [222, 148]. Our collaborators also wanted to try different prediction window sizes (e.g., predictions generated every 5 seconds vs. 15 seconds) and alternative target semantics (e.g., sitting–standing–stepping or sitting vs. not sitting). The training process also involves tuning various hyper-parameters such as learning rate and regularization coefficient.

In the above scenario it is clear that the model selection process generates dozens, if not hundreds, of different models that need to be evaluated in order to pick the best one for the prediction task. Due to the scale of the data and the complexity of the task, it is too tedious and time-consuming to manually steer this process by trying models one by one. Parallel execution on a cluster is critical for reasonable runtimes. Moreover, since our collaborators often changed the time windows and output semantics for health-related analyses, we had to rerun the whole model selection process over and over several times to get the best accuracy for their evolving task definitions. Finally, reproducible model training is also a key requirement in such scientific settings. All this underscores the importance of automatically scaling deep net model selection on a cluster with high throughput.

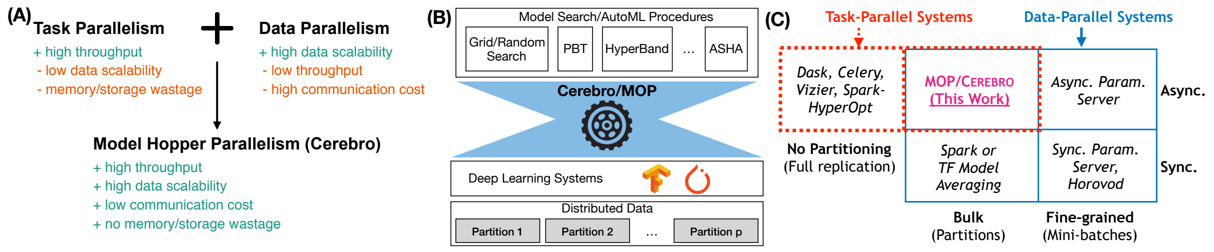


Figure 3.1. (A) Cerebro combines the advantages of both task- and data-parallelism. (B) System design philosophy and approach of CEREBRO/MOP (introduced in [212]): "narrow waist" architecture in which multiple model selection procedures and multiple deep learning tools are supported–unmodified–for specifying/executing deep net computations. MOP is our novel resource-efficient distributed SGD execution approach. (C) Model Hopper Parallelism (MOP) as a hybrid approach of task- and data-parallelism. It is the first known form of *bulk asynchronous* parallelism, filling a major gap in the parallel data systems literature.

System Desiderata. We have the following key desiderata for a deep net model selection system:

1. **Scalability.** Deep learning often has large training datasets, larger than single-node memory and sometimes even disk. Deep net model selection is also highly compute-

intensive. Thus, we desire out-of-the-box scalability to a cluster with large partitioned datasets (*data scalability*) and distributed execution (*compute scalability*).

2. **High Throughput.** Regardless of manual grid/random searches or AutoML searches, a key bottleneck for model selection is *throughput*: how many training configurations are evaluated per unit time. Higher throughput enables ML users to iterate through more configurations in bulk, potentially reaching a better accuracy sooner.

3. **Overall Resource Efficiency.** Deep net training uses variants of mini-batch stochastic gradient descent (SGD) [43, 49, 51]. To improve efficiency, the model selection system has to *avoid wasting resources* and *maximize resource utilization* for executing SGD on a cluster. We have 4 key components of resource efficiency: (1) *per-epoch efficiency*: time to complete an epoch of training; (2) *convergence efficiency*: time to reach a given accuracy metric; (3) *memory/storage efficiency*: amount of memory/storage used by the system; and (4) *communication efficiency*: amount of network bandwidth used by the system. In cloud settings, compute, memory/storage, and network all matter for overall costs because resources are pay-as-you-go; on shared clusters, which are common in academia, wastefully hogging any resource is unethical.

4. **Reproducibility.** Ad hoc model selection with distributed training is a key reason for the "reproducibility crisis" in deep learning [290]. While some Web giants may not care about unreproducibility for some use cases, this is a showstopper issue for many enterprises due to auditing, regulations, and/or other legal reasons. Most domain scientists also inherently value reproducibility.

Limitations of Existing Landscape. We compared existing approaches to see how well they cover the above desiderata. Unfortunately, each approach falls short on some major desiderata, as we summarize next. Figure 3.3 and Section 3.2 present our analysis in depth.

1. **False Dichotomy of Task- and Data-Parallelism.** Prior work on model selection systems, primarily from the ML world, almost exclusively focus on the task-parallel setting [176, 175, 130]. This ignores a pervasive approach to scale to large data on clusters: data partitioning (sharding). A disjoint line of work on data-parallel ML systems do consider partitioned data but focus on training one model at a time, not model selection workloads [257, 177]. Model selection on partitioned datasets is important because parallel file systems (e.g., HDFS for Spark), parallel RDBMSs, and "data lakes" typically store large datasets in that manner.

2. **Resource Inefficiencies.** Due to the false dichotomy, naively combining the above mentioned approaches could cause overheads and resource wastage (Section 3.2 explains more). For instance, using task-parallelism on HDFS requires extra data movement and potential caching, substantially wasting network and memory/storage resources. An alternative is remote data storage (e.g., S3) and reading repeatedly at every iteration of SGD. But this leads to orders of magnitude higher network costs by flooding the network with lots of redundant data reads. On the other hand, data-parallel systems that train one model at a time (e.g., Horovod [257] and Parameter Servers [177]) incur high communication costs, leading to high runtimes.

Overall, we see a major gap between task- and data-parallel systems today, which leads to substantially lower *overall resource efficiency, i.e., when compute, memory/storage, and network are considered holistically*.

Our Proposed System We present CEREBRO, a new system for deep learning model selection that mitigates the above issues with both task- and data-parallel execution. As Figure 3.1(A) shows, CEREBRO combines the advantages of both task- and data-parallelism, while avoiding the limitations of each. It raises model selection throughput without raising resource costs. Our target setting is *small clusters* (say, tens of nodes), which covers a vast majority (over 90%) of parallel ML workloads in practice [228]. We focus on the common setting of partitioned

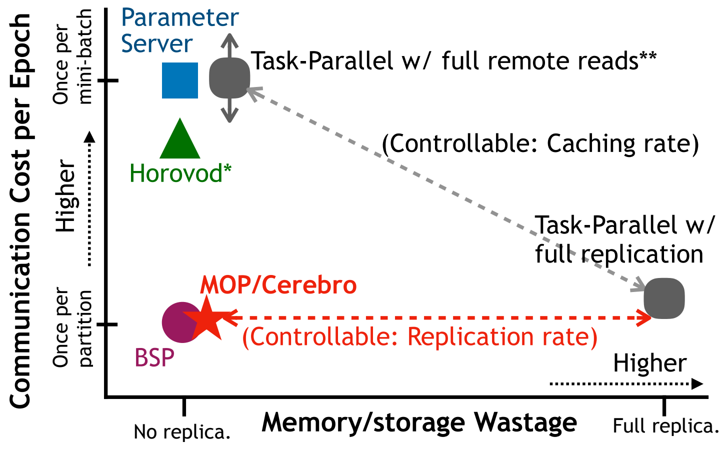

Communication Cost per Epoch
Once per mini-batch
Higher
Once per partition
Parameter Server
Task-Parallel w/ full remote reads**
(Controllable: Caching rate)
Horovod*
Task-Parallel w/ full replication
MOP/Cerebro
BSP
(Controllable: Replication rate)
Higher
No replica.
Memory/storage Wastage
Full replica.

Figure 3.2. Conceptual comparison of MOP/CEREBRO with prior art on two key axes of resource efficiency: communication cost per epoch and memory/storage wastage. Dashed line means that approach has a controllable parameter. *Horovod uses a more efficient communication mechanism than Parameter Server (PS), leading to a relatively lower communication cost. **Task-Parallelism with full remote reads has varying communication costs (higher or lower than PS) based on dataset size.

data on such clusters. Figure 3.1(B) shows the system design philosophy of CEREBRO: a narrow-waist architecture inspired by [160] to support multiple AutoML procedures and deep net frameworks.

Summary of Our Techniques. At the heart of CEREBRO is a simple but novel hybrid of task- and data-parallelism we call *model hopper parallelism* (MOP) that fulfills all of our desiderata. MOP is based on our insight about a formal optimization theoretic property of SGD: *robustness to the random ordering of the data*. Figure 3.1(C) positions MOP against prior approaches: it is the first known form of "Bulk Asynchronous" parallelism, a hybridization of the Bulk Synchronous parallelism common in the database world and task-parallelism common in the ML world. As Figure 3.2 shows, MOP has the network and memory/storage efficiency of BSP but offers much better ML convergence behavior. Prior work has shown that the BSP approach for distributed SGD (also called "model averaging") has poor convergence behavior [90]. Overall, *considering all resources holistically–compute, memory/storage, and network–MOP can be the resource-optimal choice* in our target setting.

With MOP as its basis, CEREBRO devises an *optimizing scheduler* to efficiently execute deep net model selection on small clusters. We formalize our scheduling problem as a mixed integer linear program (MILP). We compare alternate candidate algorithms with simulations and find that a simple randomized algorithm has surprisingly good performance on all aspects (Section 3.5). We then extend our scheduler to support replication of partitions, fault tolerance, and elasticity out of the box (Sections 3.5.5 and 3.5.6). Such systems-level features are crucial for deep net model selection workloads, which can often run for days. We also weigh a hybrid of CEREBRO with Horovod for model selection workloads with low degrees of parallelism. Overall, this paper makes the following contributions:

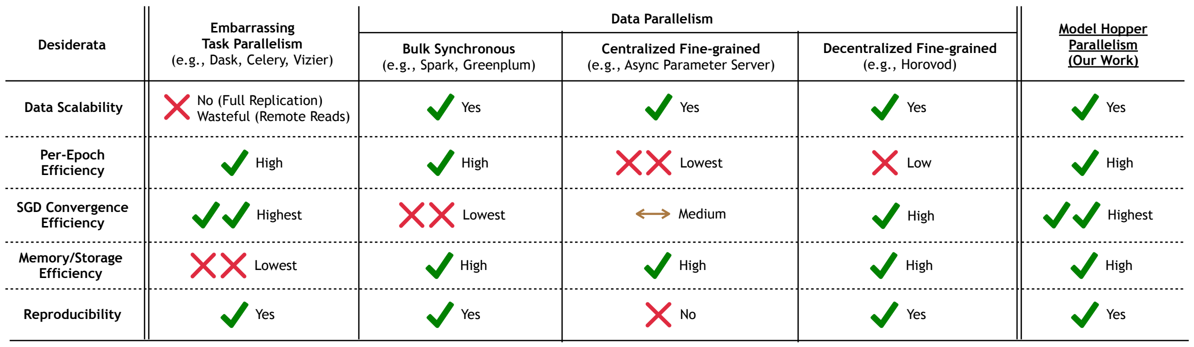
- We present a new parallel SGD execution approach we call *model hopper parallelism* (MOP) that satisfies all the desiderata listed earlier by exploiting a formal property of SGD. MOP is applicable to *any* ML models trained with SGD. We focus primarily on deep nets due to their growing popularity combined with the pressing issue of their resource-intensiveness.

- We build CEREBRO, a general and extensible deep net model selection system using MOP. CEREBRO can support arbitrary deep nets and data types, as well as multiple deep learning tools and AutoML procedures. We integrate it with TensorFlow and PyTorch.

- We formalize the scheduling problem of CEREBRO and compare 3 alternatives (MILP solver, approximate, and randomized) using simulations. We find that a randomized scheduler works well in our setting.

- We extend CEREBRO to exploit partial data replication and also support fault tolerance and elasticity.

- We perform extensive experiments on real model selection workloads with two large benchmark ML datasets: *ImageNet* and *Criteo*. CEREBRO offers 3x to 10x runtime gains over purely data-parallel systems and up to 8x memory/storage gains over purely task-parallel systems. CEREBRO also exhibits linear speedup behavior.



Desiderata	Embarrassing Task Parallelism (e.g., Dask, Celery, Vizier)	Data Parallelism			Model Hopper Parallelism (Our Work)
		Bulk Synchronous (e.g., Spark, Greenplum)	Centralized Fine-grained (e.g., Async Parameter Server)	Decentralized Fine-grained (e.g., Horovod)	
Data Scalability	✗ No (Full Replication) Wasteful (Remote Reads)	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Per-Epoch Efficiency	✓ High	✓ High	✗✗ Lowest	✗ Low	✓ High
SGD Convergence Efficiency	✓✓ Highest	✗✗ Lowest	↔ Medium	✓ High	✓✓ Highest
Memory/Storage Efficiency	✗✗ Lowest	✓ High	✓ High	✓ High	✓ High
Reproducibility	✓ Yes	✓ Yes	✗ No	✓ Yes	✓ Yes

Figure 3.3. Qualitative comparisons of existing systems on key desiderata for a model selection system.

3.2 Prior Art for Distributed Deep Learning Training

Most deep learning tools (e.g., TensorFlow) focus on the latency of training *one model at a time*, not on throughput. A popular way to raise throughput is *parallelism*. Thus, various multi-node parallel execution approaches have been studied. All of them fall short on some desiderata, as Figure 3.3 shows. We group these approaches into 4 categories:

Embarrassingly Task Parallel. Tools such as Python Dask, Celery, Vizier [104], and Ray [206] can run different training configurations on different workers in a task-parallel manner. Each worker can use logically sequential SGD, which yields the best convergence efficiency. This is also reproducible. There is no communication across workers during training, but the whole dataset must be copied to each worker, which does not scale to large partitioned datasets. Copying datasets to all workers is also *highly wasteful of resources*, both memory and storage, which raises costs. Alternatively, one can use remote storage (e.g., S3) and read data remotely every epoch. But such repeated reads wastefully flood the network with orders of magnitude extra redundant data, e.g., see a realistic cost calculation in Table 3.2..

Bulk Synchronous Parallel (BSP). BSP systems such as Spark and TensorFlow with model averaging [17] parallelize one model at a time. They partition the dataset across workers, yielding high memory/storage efficiency. They broadcast a model, train models independently on each worker's partition, collect all models on the master, average the weights (or gradients), and repeat this every epoch. Alas, this approach converges poorly for highly non-convex models; so, it is almost never used for deep net training [263].

Centralized Fine-grained. These systems also parallelize one model at a time on partitioned data but at the finer granularity of each mini-batch. The most prominent example is Parameter Server (PS) [177]. PS is a set of systems for data-parallel ML. A typical PS consists of *servers* and *workers*; servers maintain the globally shared model weights, while workers compute SGD gradients on a locally stored data partition. Workers communicate with servers periodically to update and retrieve model weights. Based on the nature of these communications, PS has two variants: *synchronous* and *asynchronous*. Asynchronous PS is highly scalable but unreproducible; it often has poorer convergence than synchronous PS due to stale updates but synchronous PS has higher overhead for synchronization.

All PS-style approaches have *high communication* due to their centralized all-to-one communications, which is proportional to the number of mini-batches and orders of magnitude higher than BSP, e.g., 10,000x in Table 3.2.

Decentralized Fine-grained. The best example is Horovod [257]. It adopts HPC-style techniques to enable synchronous all-reduce SGD. While this approach is bandwidth optimal, communication latency is still proportional to the number of workers, and the synchronization barrier can become a bottleneck. The total communication overhead is also proportional to the number of mini-batches and orders of magnitude higher than BSP, e.g., 9,000x in Table 3.2.

Table 3.1. Notation used in Section 3.3

Symbol	Description
S	Set of training configurations
p	Number of data partitions/workers
k	Number of epochs for S to be trained
m	Model size (uniform for exposition sake)
b	Mini-batch size
D	Training dataset ($\langle D \rangle$: dataset size, $\lvert D \rvert$: number of examples)

3.3 Model Hopper Parallelism

We first explain how MOP works and its properties. Table 3.1 presents some notation. We also theoretically compare the communication costs of MOP and prior approaches.

3.3.1 Basic Idea of MOP

We are given a set S of training configurations ("configs" for short). For simplicity of exposition, assume for now each runs for k epochs–we relax this later[1]. Shuffle the dataset once and split into p partitions, with each partition located on one of p worker machines. Given these inputs, MOP works as follows. Pick p configs from S and assign one per worker (Section 3.5 explains how we pick the subset). On each worker, the assigned config is trained on the local partition for a single *sub-epoch*, which we also call a *training unit*. Completing a training unit puts that worker back to the idle state. An idle worker is then assigned a new config that has not already been trained and also not being currently trained on another worker. Overall, a model "hops" from one worker to another after a sub-epoch. Repeat this process until all configs are trained on all partitions, completing one epoch for each model. Repeat this every epoch until all configs in S are trained for k epochs. The invariants of MOP can be summarized as follows:

[1]Section 4.2 (Supporting Multiple AutoML Procedures) explains further how CEREBRO can support different configs being trained for different numbers of epochs.

- *Completeness:* In a single epoch, each training config is trained on all workers exactly once.
- *Model training isolation:* Two training units of the same config are not run simultaneously.
- *Worker/partition exclusive access:* A worker executes only one training unit at a time.
- *Non-preemptive execution:* An individual training unit is run without preemption once started.

Insights Underpinning MOP. MOP exploits a formal property of SGD: *any random ordering* of examples suffices for convergence [43, 49]. Each of the p configs visits the data partitions in a different (pseudorandom) yet in sequential order. Thus, MOP offers high accuracy for all models, comparable to sequential SGD. While SGD's robustness has been exploited before in ML systems, e.g., in Parameter Server [177], MOP exploits it at the *partition level* instead of at the mini-batch level to reduce communication costs. This is possible because we connect this property with model selection workloads instead of training one model at a time.

Positioning MOP. As Figure 3.1(C) shows, MOP is a new hybrid of task- and data-parallelism that is a form of "bulk asynchronous" parallelism. Like task-parallelism, MOP trains many configs in parallel but like BSP, it runs on partitions. So, MOP is more fine-grained than task parallelism but more coarse-grained than BSP. MOP has no global synchronization barrier within an epoch. Later in Section 3.5, we dive into how CEREBRO uses MOP to schedule S efficiently and in a general way. Overall, while the core idea of MOP is simple–perhaps even obvious in hindsight–it has hitherto not been exploited in its full generality in ML systems.

Reproducibility. MOP does not restrict the visit ordering. So, reproducibility is trivial in MOP: log the worker visit order for each configuration per epoch and replay with this order. Crucially, this logging incurs very negligible overhead because a model hops only *once per partition*, not for every mini-batch, at each epoch.

Table 3.2. Communication cost analysis of MOP and other approaches. *Full replication. †Remote reads. ‡Parameters for the example: $k = 20$, $|S| = 20$, $p = 10$, $m = 1$GB, $\langle D \rangle = 1$TB, and $|D|/b = 100$K.

	Comm. Cost	**Example**‡
Model Hopper Parallelism	$kmp\|S\| + m\|S\|$	4 TB
Task Parallelism (FR*)	$p\langle D \rangle + m\|S\|$	10 TB
Task Parallelism (RR†)	$k\|S\|\langle D \rangle + m\|S\|$	400 TB
Bulk Synchronous Parallelism	$2kmp\|S\|$	8 TB
Centralized Fine-grained	$2kmp\|S\|\left\lceil \frac{\|D\|}{bp} \right\rceil$	80 PB
Decentralized Fine-grained	$2km(p-1)\|S\|\left\lceil \frac{\|D\|}{bp} \right\rceil$	72 PB

3.3.2 Communication Cost Analysis

We summarize the communication costs of MOP and other approaches in Table 3.2. It also illustrates the communication costs in bytes for a realistic example based on our case study in Section 3.1. MOP reaches the theoretical minimum cost of $kmp|S|$. Crucially, note that this cost does not depend on batch size, which underpins MOP's higher efficiency. BSP also has the same asymptotic cost but unlike MOP, BSP typically converges poorly for deep nets and lacks sequential-equivalence. Fine-grained approaches like PS and Horovod have communication costs proportional to the number of mini-batches, which can be orders of magnitude higher. In our setting, p is under low 10s, but the number of mini-batches can even be 1000s to millions based on the batch size.

3.4 System Overview

We present an overview of CEREBRO, an ML system that uses MOP to execute deep net model selection workloads.

3.4.1 User-facing API

CEREBRO API allows users to do 2 things: (1) register workers and data; and (2) issue a deep net model selection workload. Workers are registered by IP addresses. As for datasets, CEREBRO expects a list of data partitions and their availability on each worker. We assume shuffling and partitioning are already handled by other means, since these are well studied. This common data ETL step is also orthogonal to our focus and is not a major part of the total runtime for iterative deep net training.

CEREBRO takes the reference to the dataset, set of initial training configs, the AutoML procedure, and 3 user-defined functions: *input_fn*, *model_fn*, and *train_fn*. It first invokes *input_fn* to read and pre-process the data. It then invokes *model_fn* to instantiate the neural architecture and potentially *restore* the model state from a previous *checkpointed* state. The *train_fn* is invoked to perform one sub-epoch of training. We assume validation data is also partitioned and use the same infrastructure for evaluation. During evaluation, CEREBRO marks model parameters as non-trainable before invoking *train_fn*. We also support higher-level API methods for AutoML procedures that resemble the popular APIs of Keras [227]. Note that *model_fn* is highly general, i.e., CEREBRO supports *all* neural computational graphs on all data types supported by the underlying deep learning tool, including CNNs, RNNs, transformers, etc. on structured data, text, images, video, etc. Due to space constraints, more details of our APIs, including full method signatures and a fleshed out example of how to use CEREBRO are provided in the appendix of our technical report [317].

3.4.2 System Architecture

We adopt an extensible architecture, as Figure 3.4 shows. This allows us to easily support multiple deep learning tools and AutoML procedures. There are 5 main components: (1) API, (2) Scheduler, (3) Task Executor, (4) Catalog, and (5) Resource Monitor. Scheduler is responsible for orchestrating the entire workload. It relies on worker and data availability information from

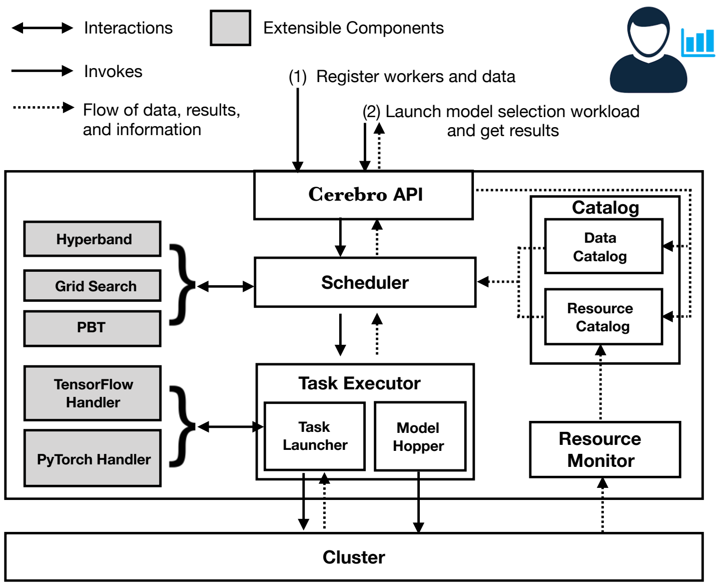
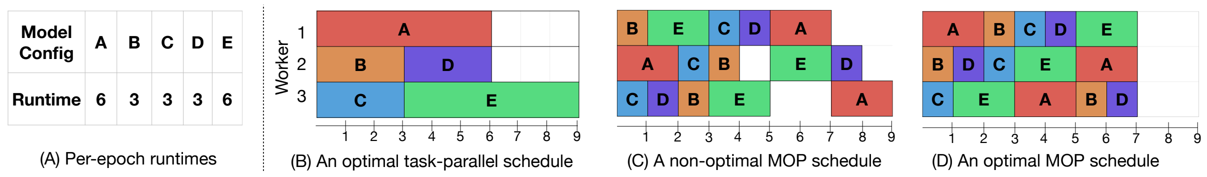


Figure 3.4. System architecture of CEREBRO.

the Catalog. Task Executor launches training units on the cluster and also handles model hops. Resource Monitor is responsible for detecting worker failures and updating the Resource Catalog. Section 3.5 explains how the Scheduler works and how we achieve fault tolerance and elasticity. Next, we describe how CEREBRO's architecture enables high system generality.



Model Config	A	B	C	D	E
Runtime	6	3	3	3	6

Figure 3.5. Gantt charts of task-parallel and MOP schedules for a sample model selection workload.

Supporting Multiple Deep Learning Tools. The functions *input_fn*, *model_fn*, and *train_fn* are written by users in the deep learning tool's APIs. We currently support TensorFlow and PyTorch (it is simple to add support for more). To support multiple such tools, we adopt a handler-based architecture to delineate tool-specific aspects: model training, checkpointing and restoring. Note that checkpointing and restoring is how CEREBRO realizes model hops. Task Executor automatically injects the tool-specific aspects from the corresponding tool's handler and runs these functions on the workers. Overall, CEREBRO's architecture is highly general and supports virtually all forms of data types, deep net architectures, loss functions, and SGD-based optimizers.

Supporting Multiple AutoML Procedures. Metaheuristics called AutoML procedures are common for exploring training configs. We now make a key observation about such procedures that underpins our Scheduler. Most AutoML procedures fit a *common template*: create an initial set of configs (S) and evaluate them after each epoch (or every few epochs). Based on the evaluations, terminate some configurations (e.g., as in Hyperband [176] and PBT [130]) or add new configurations (e.g., as in PBT). Grid/random search is a one-shot instance of this template. Thus, we adopt this template for our Scheduler. Given S, CEREBRO trains all models in S for one epoch and passes control back to the corresponding AutoML procedure for convergence/termination/addition evaluations. CEREBRO then gets a potentially modified set S' for the next epoch. This approach also lets CEREBRO support data re-shuffling after each epoch. But the default (and common practice) is to shuffle only once upfront. Grid/random search (perhaps the most popular in practice), Hyperband, and PBT (and more procedures) conform to this common template and are currently supported.

ASHA [175] and Hyperopt [41] are two notable exceptions to the above template, since they do not have a global synchronized evaluation of training configs after an epoch and are somewhat tied to task-parallel execution. While MOP/CEREBRO cannot ensure logically same execution as ASHA or HyperOpt on task-parallelism, it is still possible to emulate them on MOP/CEREBRO without any modifications to our system. In fact, our experiments with ASHA show that ASHA on CEREBRO has comparable–even slightly better!–convergence behavior than ASHA on pure task-parallelism (Section 3.6.3).

3.4.3 System Implementation Details

We prototype CEREBRO in Python using XML-RPC client-server package. Scheduler runs on the client. Each worker runs a single service. Scheduling follows a push-based model–

Table 3.3. Additional notation used in the MOP MILP formulation

Symbol	Description
$T \in \mathbb{R}^{\|S\| \times p}$	$T_{i,j}$ is the runtime of unit $s_{i,j}$ (i^{th} configuration on j^{th} worker)
C	Makespan of the workload
$X \in \mathbb{R}^{\|S\| \times p}$	$X_{i,j}$ is the start time of the execution of i^{th} configuration on j^{th} partition/-worker
$Y \in \{0,1\}^{\|S\| \times p \times p}$	$Y_{i,j,j'} = 1 \iff X_{i,j} < X_{i,j'}$
$Z \in \{0,1\}^{\|S\| \times \|S\| \times p}$	$Z_{i,i',j} = 1 \iff X_{i,j} < X_{i',j}$
V	Very large value (Default: sum of training unit runtimes)

Scheduler assigns tasks and periodically checks the responses from the workers. We use a shared network file system (NFS) as the central repository for models. Model hopping is realized implicitly by workers writing models to and reading models from this shared file system. Technically, this doubles the communication cost of MOP to $2kmp|S|$, still a negligible overhead. Using NFS greatly reduces engineering complexity to implement model hops.

3.5 Cerebro Scheduler

Scheduling training units on workers properly is critical because pathological orderings can under-utilize resources substantially, especially when deep net architectures and/or workers are heterogeneous. Consider the model selection workload shown in Figure 3.5(A). Assume workers are homogeneous and there is no data replication. For one epoch of training, Figure 3.5(B) shows an optimal task-parallel schedule for this workload with a 9-unit makespan. Figure 3.5(C) shows a non-optimal MOP schedulewith also 9 units makespan. But as Figure 3.5(D) shows, an optimal MOP schedule has a makespan of only 7 units. Overall, we see that MOP's training unit-based scheduling offers more flexibility to raise resource utilization. Next, we formally define the MOP-based scheduling problem and explain how we design our Scheduler.

3.5.1 Formal Problem Statement as MILP

Suppose the runtimes of each training unit, aka *unit times*, are given. These can be obtained with, say, a pilot run for a few mini-batches and then extrapolating (this overhead will be marginal). For starters, assume each of the p data partitions is assigned to only one worker. The objective and constraints of the MOP-based scheduling problem is as follows. Table 3.3 lists the additional notation used here.

$$\texttt{Objective:} \quad \min_{C,X,Y,Z} C \tag{3.1}$$

`Constraints:`

$$\begin{aligned}
&\forall i,i' \in [1,\ldots,|S|]\ \forall j,j' \in [1,\ldots,p]\\
&(a)\ X_{i,j} \geq X_{i,j'} + T_{i,j'} - V \cdot Y_{i,j,j'}\\
&(b)\ X_{i,j'} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Y_{i,j,j'})\\
&(c)\ X_{i,j} \geq X_{i',j} + T_{i',j} - V \cdot Z_{i,i',j}\\
&(d)\ X_{i',j} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Z_{i,i',j})\\
&(e)\ X_{i,j} \geq 0\\
&(f)\ C \geq X_{i,j} + T_{i,j}
\end{aligned} \tag{3.2}$$

We need to minimize makespan C, subject to the constraints on C, unit start times X, model training isolation matrix Y, and worker/partition exclusive access matrix Z. The constraints enforce some of the invariants of MOP listed in Section 3.3. Equations 2.a and 2.b ensure model training isolation. Equations 2.c and 2.d ensure worker exclusive access. Equation 2.e ensures that training unit start times are non-negative and Equation 2.f ensures that C captures the time taken to complete all training units.

Given the above, a straightforward approach to scheduling is to use an MILP solver like Gurobi [112]. The start times X then yield the actual schedule. But our problem is essentially an instance of the classical open-shop scheduling problem, which is known to be `NP-Hard` [107]. Since $|S|$ can even be 100s, MILP solvers may be too slow (more in Section 3.5.4); thus, we explore alternative approaches.

3.5.2 Approximate Algorithm-based Scheduler

For many special cases, there are algorithms with good approximation guarantees that can even be optimal under some conditions. One such algorithm is "vector rearrangement" [293, 94]. It produces an optimal solution when $|S| \gg p$, which is possible in our setting.

The vector rearrangement based method depends on two values: L_{max} (see Equation 3.3), the maximum load on any worker; and T_{max} (see Equation 3.4), the maximum unit time of any training configuration in S.

$$L_{max} = \max_{j \in [1,\ldots,p]} \sum_{i=1}^{|S|} T_{i,j} \tag{3.3}$$

$$T_{max} = \max_{i \in [1,\ldots,|S|], j \in [1,\ldots,p]} T_{i,j} \tag{3.4}$$

If $L_{max} \geq (p^2 + p - 1) \cdot T_{max}$, this algorithm's output is optimal. When there are lots of configs, the chance of the above constraint being satisfied is high, yielding us an optimal schedule. But if the condition is not met, the schedule produced yields a makespan $C \leq C^* + (p-1) \cdot T_{max}$, where C^* is the optimal makespan value. This algorithm scales to large $|S|$ and p because it runs in polynomial time in contrast to the MILP solver. For more details on this algorithm, we refer the interested reader to [293, 94].

3.5.3 Randomized Algorithm-based Scheduler

The approximate algorithm is complex to implement in some cases, and its optimality condition may be violated often. Thus, we now consider a much simpler scheduler based on

Algorithm 1. Randomized Scheduling

```
1: Input: $S$
2: $Q = \{s_{i,j} : \forall i \in [1,\dots,|S|], \forall j \in [1,\dots,p]\}$
3: worker_idle $\leftarrow$ [true,...,true]
4: model_idle $\leftarrow$ [true,...,true]
5: while not empty($Q$) do
6:   for $j \in [1,\dots,p]$ do
7:     if worker_idle[$j$] then
8:       $Q \leftarrow$ shuffle($Q$)
9:       for $s_{i,j'} \in Q$ do
10:        if model_idle[$i$] and $j' = j$ then
11:          Execute $s_{i,j'}$ on worker $j$
12:          model_idle[$i$] $\leftarrow$ false
13:          worker_idle[$j$] $\leftarrow$ false
14:          remove($Q, s_{i,j'}$)
15:          break
16:   wait WAIT_TIME
```

Algorithm 2. When $s_{i,j}$ finishes on worker j

```
1: model_idle[$i$] $\leftarrow$ true
2: worker_idle[$j$] $\leftarrow$ true
```

randomization. This approach is simple to implement and offer much more flexibility (explained more later). Algorithm 1 presents our randomized scheduler.

Given S, create $Q = \{s_{i,j} : \forall i \in [1,...,|S|], j \in [1,..,p]\}$, the set of all training units. Note that $s_{i,j}$ is the training unit of configuration i on worker j. Initialize the state of all models and workers to idle state. Then find an idle worker and schedule a random training unit from Q on it. This training unit must be such that its configuration is not scheduled on another worker and it corresponds to the data partition placed on that worker (Line 10). Then remove the chosen training unit from Q. Continue this process until no worker is idle and eventually, until Q is empty. After a worker completes training unit $s_{i,j}$ mark its model i and worker j as idle again as per Algorithm 2.

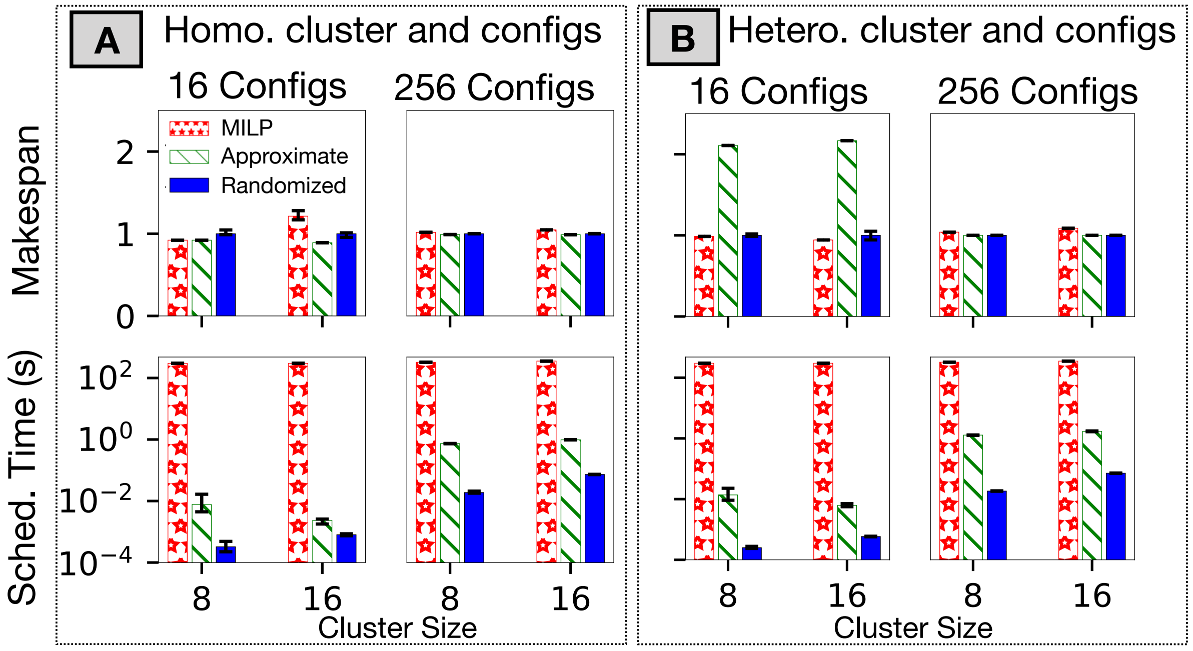


Figure 3.6. Scheduler runtimes and makespans of the schedules produced in different settings. Makespans are normalized with respect to that of Randomized. (A) Homogeneous cluster and homogeneous training configs. (B) Heterogeneous cluster and heterogeneous training configs.

3.5.4 Comparing Different Scheduling Methods

We use simulations to compare the efficiency and makespans yielded by the three alternative schedulers. The MILP and approximate algorithm are implemented using Gurobi. We set a maximum optimization time of 5min for tractability sake. We compare the scheduling methods on 3 dimensions: 1) number of training configs (two values: 16 and 256), 2) number of workers (two values: 8 and 16), 3) homogeneity/heterogeneity of configs and workers.

Sub-epoch training time (unit time) of a training config is directly proportional to the compute cost of the config and inversely proportional to compute capacity of the worker. For the homogeneous setting, we initialize all training config compute costs to be the same and also all worker compute capacities to be the same. For the heterogeneous setting, training config compute costs are randomly sampled (with replacement) from a set of popular deep CNNs (n=35) obtained from [27]. The costs vary from 360 MFLOPS to 21000 MFLOPS with a mean of 5939 MFLOPS and standard deviation of 5671 MFLOPS. Due to space constraints we provide these computational costs in the Appendix of our technical report [317]. For worker compute capacities, we randomly sample (with replacement) compute capacities from 4 popular Nvidia GPUs: Titan Xp (12.1 TFLOPS/s), K80 (5.6 TFLOPS/s), GTX 1080 (11.3 TFLOPS/s), and P100 (18.7 TFLOPS/s). For each setting, we report the average of 5 runs with different random seeds set to the scheduling algorithms and also the min and max of all 5 runs. All makespans reported are normalized by the randomized scheduler's makespan.

The MILP scheduler sometimes performs poorer than the other two because it has not converged to the optimal in the given time budget. The approximate scheduler performs poorly when both the configs and workers are heterogeneous. It is also slower than the randomized scheduler.

Overall, the randomized approach works surprisingly well on all aspects: near-optimal makespans with minimal variance across runs and very fast scheduling. We believe this interesting superiority of the randomized algorithm against the approximation algorithm is due to some fundamental characteristics of deep net model selection workloads, e.g., large number of configurations and relatively low differences in compute capacities. We leave a thorough theoretical analysis of the randomized algorithm to future work. Based on these results, we use the randomized approach as the default Scheduler in CEREBRO.

3.5.5 Replica-Aware Scheduling

So far we assumed that a partition is available on only one worker. But some file systems (e.g., HDFS) often replicate data files, say, for reliability sake. We now exploit such replicas for more scheduling flexibility and faster plans.

The replica-aware scheduler requires an additional input: availability information of partitions on workers (an availability map). In replica-aware MOP, a training configuration need *not* visit all workers. This extension goes beyond open shop scheduling, but it is still `NP-Hard` because the open shop problem is a special case of this problem with a replication factor of one. We extended the MILP scheduler but it only got slower. So, we do not use it and skip its details.

Modifying the approximate algorithm is also non-trivial because it is tightly coupled to the open shop problem; so, we skip that too. In contrast, the randomized scheduler can be easily extended for replica-aware scheduling. The only change needed to Algorithm 1 is in Line 10: instead of checking $j' = j$, consult the availability map to check if the relevant partition is available on that worker.

3.5.6 Fault Tolerance and Elasticity

We now explain how we make our randomized scheduler fault tolerant. Instead of just Q, we maintain two data structures Q and Q'. Q' is initialized to be empty. The process in Algorithm 1 continues until both Q and Q' are empty. When a training unit is scheduled, it will be removed from Q as before but now also *added* to Q'. It will be removed from Q' when it successfully completes its training on the assigned worker. But if the worker fails before the training unit finishes, it will be moved back from Q' to Q. If the data partitions present on the failed worker are also available elsewhere, the scheduler will successfully execute the corresponding training units on those workers at a future iteration of the loop in Algorithm 1.

CEREBRO detects failures via the periodic heart-beat check between the scheduler and workers. Because the trained model states are always checkpointed between training units, they can be recovered and the failed training units can be restarted. Only the very last checkpointed model is needed for the failure recovery and others can be safely deleted for reclaiming storage. The same mechanism can be used to detect availability of new compute resources and support seamless scale-out elasticity in CEREBRO.

3.5.7 Extension: Horovod Hybrid

Some AutoML procedures (e.g., Hyperband) start with large $|S|$ but then kill some non-promising configs after some epochs. So, only a few configs may train till convergence. This means at the later stages, we may encounter a situation where $|S|$ goes below p. In such cases, CEREBRO can under-utilize the cluster. To overcome this limitation, we explored the possibility

Table 3.4. Dataset details. All numbers are after preprocessing and sampling of the datasets.

Dataset	On-disk size	Count	Format	Class
ImageNet	250 GB	1.2M	HDF5	1000
Criteo	400 GB	100M	TFRecords	Binary

of doubly hybridizing MOP with data-parallelism by implementing a hybrid of CEREBRO and Horovod. Just like CEREBRO, Horovod is also equivalent to sequential SGD; so, the hybrid is reproducible. The basic idea is simple: divide the cluster into virtual sub-clusters and run Horovod within each sub-cluster and MOP across sub-clusters. Due to space constraints, we explain this hybrid architecture further in Appendix A.

3.6 Experimental Evaluation

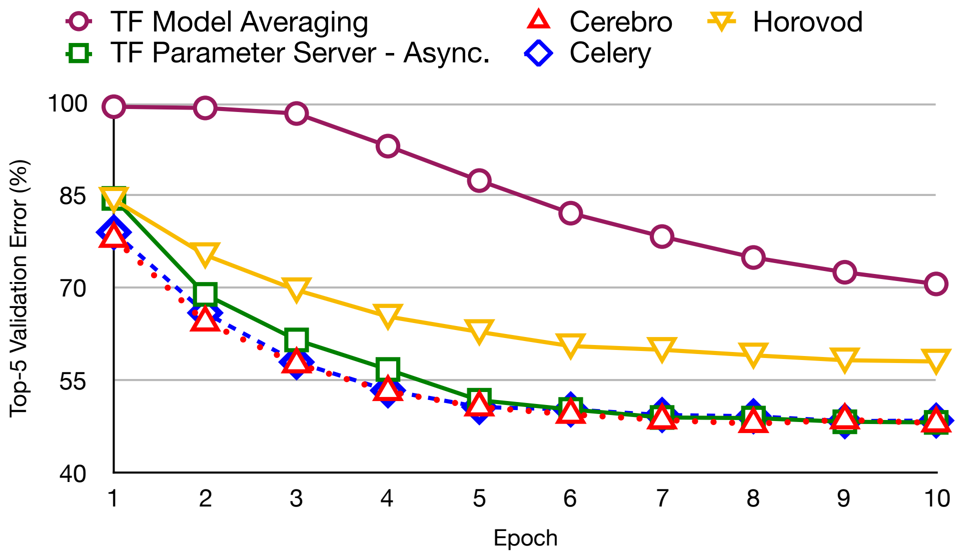
We empirically validate if CEREBRO can improve overall throughput and efficiency of deep net model selection. We then evaluate CEREBRO in depth. Finally, we demonstrate CEREBRO's ability to support multiple AutoML procedures.

Datasets. We use two large benchmark datasets: *ImageNet* [78] and *Criteo* [73]. *ImageNet* is a popular image classification dataset. We choose the 2012 version and reshape the images to 112×112 pixels[2]. *Criteo* is an ad click classification dataset with numeric and categorical features. It is shipped under sparse representation. We one-hot encode the categorical features and densify the data. Only a 2.5% random sample of the dataset is used[2]. Table 3.4. summarizes the dataset statistics.

[2]We made this decision only so that all of our experiments can complete in reasonable amount of time. This decision does *not* alter the takeaways from our experiments.

System	ImageNet			Criteo		
	Runtime (hrs)	GPU Utili. (%)	Storage Footprint (GB)	Runtime (hrs)	CPU Utili. (%)	Storage Footprint (GB)
TF PS - Async	19.00	8.6	250	28.80	6.9	400
Horovod	5.42	92.1	250	14.06	16.0	400
TF Model Averaging	1.97	72.1	250	3.84	52.2	400
Celery	1.72	82.4	2000	3.95	53.6	3200
Cerebro	1.77	79.8	250	3.40	51.9	400

(A) Per-epoch makespans and CPU/GPU utilization.



(B) Learning curves of the resp. best configs on *ImageNet*.

Figure 3.7. End-to-end results on *ImageNet* and *Criteo*. For Celery, we report the runtime corresponding to the lowest makespan schedule. Celery's per-epoch runtime varies between 1.72-2.02 hours on *ImageNet*; on *Criteo*, 3.95-5.49 hours. Horovod uses GPU kernels for communication; hence its high GPU utilization.

Workloads. For our first end-to-end test, we use two different neural architectures and grid search for hyper-parameters, yielding 16 training configs for each dataset. Table 3.5 offers

Table 3.5. Workloads.*architectures similar to VGG16 and ResNet50. †Serialized sizes.

Dataset	Model arch.	Model size/MB†	Batch size	Learning rate	Regularization	Epochs
ImageNet	{VGG16*, ResNet50*}	VGG16: 792, ResNet50: 293	{32, 256}	$\{10^{-4}, 10^{-6}\}$	$\{10^{-4}, 10^{-6}\}$	10
Criteo	3-layer NN	179	{32, 64, 256, 512}	$\{10^{-3}, 10^{-4}\}$	$\{10^{-4}, 10^{-5}\}$	5

the details. We use Adam [149] as our SGD method. To demonstrate generality, we also present results for HyperOpt and ASHA on CEREBRO in Section 3.6.3.

Experimental Setup. We use two clusters: CPU-only for *Criteo* and GPU-enabled for *ImageNet*, both on CloudLab [246]. Each cluster has 8 worker nodes and 1 master node. Each node in both clusters has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, 1TB HDD and 10 Gbps network. Each GPU cluster worker node has an extra Nvidia P100 GPU. All nodes run Ubuntu 16.04. We use TensorFlow v1.12.0 as CEREBRO's underlying deep learning tool. For GPU nodes, we use CUDA version 9.0 and cuDNN version 7.4.2. Both datasets are randomly shuffled and split into 8 equi-sized partitions.

3.6.1 End-to-End Results

We compare CEREBRO with 5 systems: 4 data-parallel–synchronous and asynchronous TensorFlow Parameter Server, Horovod, BSP-style TensorFlow model averaging–and 1 task-parallel (Celery). For Celery, we replicate datasets to each worker beforehand and stream them from disk, since they do not fit in memory. I/O time is trivial for deep nets, where computation dominates; thus, they can be interleaved. We use TensorFlow features to achieve this. For all other systems, each worker node has one in-memory data partition. We do not include data copying in the end-to-end runtimes. For scheduling, Celery uses a FIFO queue and CEREBRO uses the randomized scheduler. All other systems train models sequentially.

Figure 3.7 presents the results. CEREBRO significantly improves the efficiency and throughput of model selection. On *ImageNet*, CEREBRO is over 10x faster than asynchronous PS, which has a GPU utilization as low as 9%! Synchronous PS was even slower. CEREBRO is 3x faster than Horovod. Horovod has high GPU utilization because it uses GPU for communication. CEREBRO's runtime is comparable to model averaging, which is as expected. But note model averaging converges poorly. Celery's runtime is dependent on the execution order and thus we report the runtime on the optimal schedule. On *ImageNet*, Celery's runtime is comparable to CEREBRO. But note that Celery has a highly bloated 8x memory/storage footprint. Overall, Celery and CEREBRO have the best learning curves–this is also as expected because MOP ensures sequential equivalence for SGD, just like task-parallelism. Horovod converges slower due to its larger effective mini-batch size.

On *Criteo*, CEREBRO is 14x faster than synchronous PS and 8x faster than asynchronous PS. Both variants of PS report severe CPU under-utilization ($< 7\%$). CEREBRO is also 4x faster than Horovod. CEREBRO's runtime is comparable to model averaging, with about 52% CPU utilization. Celery is somewhat slower than CEREBRO due to a straggler issue caused by the highly heterogeneous model configs for *Criteo*. CEREBRO's MOP approach offers higher flexibility to avoid such straggler issues. A more detailed explanation is given in the appendix of our technical report [317]. All methods have almost indistinguishable convergence behavior on this dataset: all reached 99% accuracy quickly, since the class label is quite skewed.

Overall, CEREBRO is the most resource-efficient approach when compute, memory/storage, and network are considered holistically. It also has the *best accuracy behavior*, on par with task-parallelism.

3.6.2 Drill-down Experiments

Unless specified otherwise, we now show experiments on the GPU cluster, *ImageNet*, and a model selection workload of 8 configs (4 learning rates, 2 regularization values, and ResNet architectures) trained for 5 epochs. Each data partition is placed on only one worker.

Scalability. We study the speedups (strong scaling) of CEREBRO and Horovod as we vary the cluster sizes. Figure 3.8(A) shows the speedups, defined as the workload completion time on multiple workers vs a single worker. CEREBRO exhibits linear speedups due to MOP's

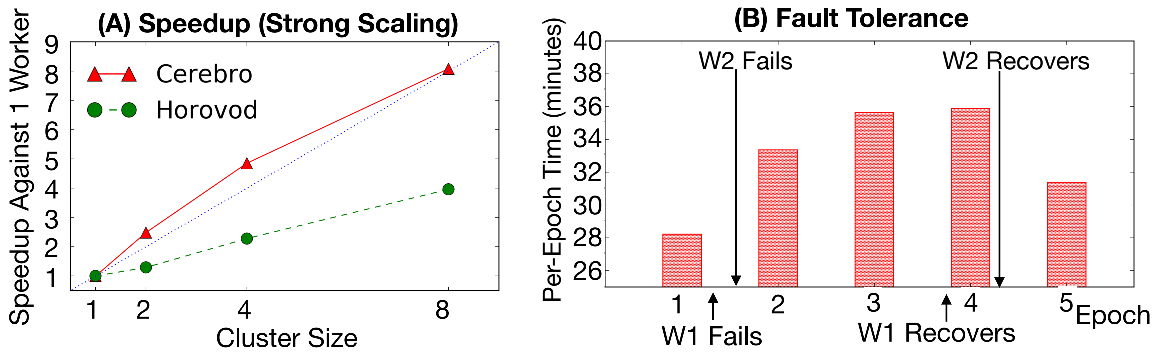


Figure 3.8. (A) Speedup plot (strong scaling). (B) Fault-tolerance.

marginal communication costs; in fact, it seems slightly super-linear here because the dataset fits entirely in cluster memory compared to the minor overhead of reading from disk on the single worker. In contrast, Horovod exhibits substantially sub-linear speedups due to its much higher communication costs with multiple workers.

Fault Tolerance. We repeat our drill-down workload with a replication factor of 3. We first inject two node failures and bring the nodes back online later. Figure 3.8(B) shows the time taken for each epoch and the points where the workers failed and returned online. Overall, we see CEREBRO's replica-aware randomized scheduler can seamlessly execute the workload despite worker failures.

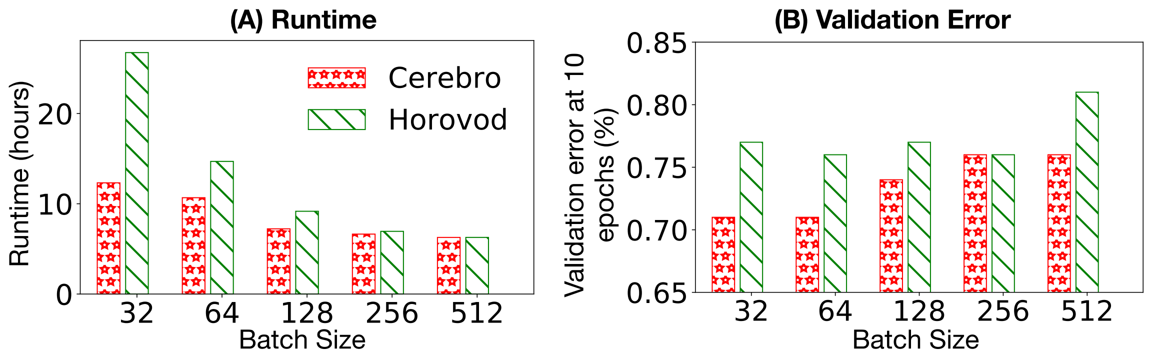


Figure 3.9. Effect of batch size on communication overheads and convergence efficiency. (A) Runtime against batch size. (B) The lowest validation error after 10 epochs against batch size.

Effect of Batch Size. We now evaluate the effect of training mini-batch size for CEREBRO and Horovod. We evaluate 5 batch sizes and report makespans and the validation error of the best model for each batch size after 10 epochs. Figure 3.9 presents the results. With batch size 32, Horovod is 2x slower than CEREBRO. However, as the batch size increases, the difference narrows since the relative communication overhead per epoch decreases. CEREBRO also runs faster with larger batch size due to better hardware utilization. The models converge slower as batch size increases. The best validation error is achieved by CEREBRO with a batch size of 32. With the same setting, Horovod's best validation error is higher than CEREBRO; this is because its effective batch size is 256 (32×8). Horovod's best validation error is closer to CEREBRO's at a batch size of 256. Overall, CEREBRO's efficiency is more stable to the batch size, since models hop per sub-epoch, not per mini-batch.

Network and Storage Efficiency. We study the tradeoff between redundant remote reads (wastes network) vs redundant data copies across workers (wastes memory/storage). Task parallelism forces users to either duplicate the dataset to all workers or store it in a common repository/distributed filesystem and read remotely. CEREBRO can avoid both forms of resource wastage. We assume the whole dataset cannot fit on single-node memory. We compare CEREBRO and Celery in the following 2 settings:

Reading from remote storage (e.g., S3). In this setting, Celery reads data from a remote storage repeatedly each epoch. For CEREBRO each worker remotely reads one data partition and caches it. We change the data scale to evaluate effects on the makespan and the amount of remote reads. Figure 3.10 shows the results. Celery is slightly slower than CEREBRO due to remote read overheads. The most significant advantage of CEREBRO is its network bandwidth cost, which is over 10x lower than Celery's. After the initial read, CEREBRO only communicates models weights during training. In situations where reads and networks are not free (e.g., cloud providers), Celery will incur higher monetary costs than CEREBRO. These results show it is perhaps better to partition the dataset on S3, cache partitions on workers on the first read, and then run CEREBRO instead of Celery with full dataset reads from S3 per epoch to avoid copying.

Reading from distributed storage (e.g., HDFS). In this setting, the dataset is partitioned, replicated, and stored on 8 workers. We then load all local data partitions into each worker's

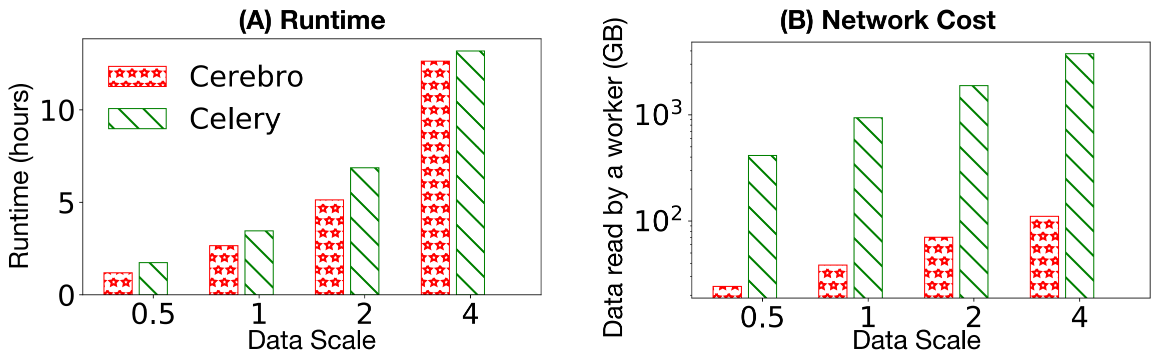


Figure 3.10. Reading data from remote storage.

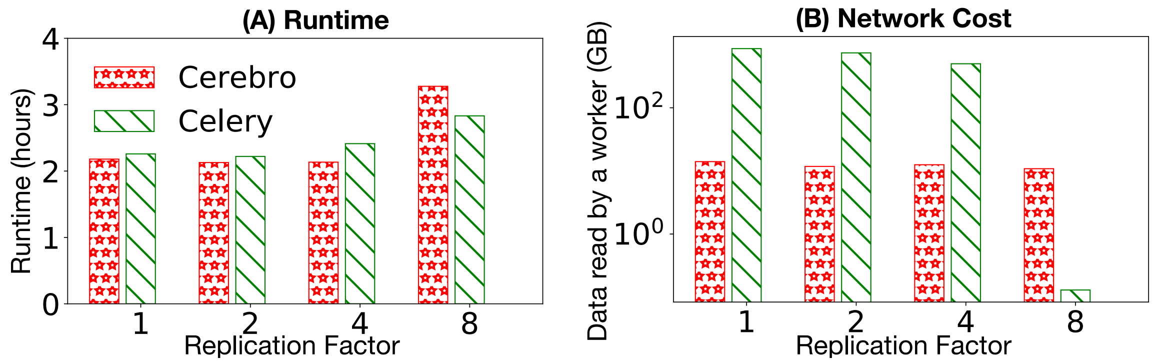


Figure 3.11. Reading data from distributed storage.

memory. Celery performs remote reads for non-local partitions. We vary the replication factor to study its effect on the makespan and the number of remote reads. Figure 3.10 presents the results. For replication factors 1 (no replication), 2, and 4, CEREBRO incurs 100x less network usage and is slightly faster than Celery. But at a replication factor of 8 (i.e., full replication), CEREBRO is slightly slower due to the overhead of model hops. For the same reason, CEREBRO incurs marginal network usage, while Celery has almost no network usage other than control actions. Note that the higher the replication factor for Celery, the more memory/storage is wasted. CEREBRO offers the best overall resource efficiency–compute, memory/storage, and network put together–for deep net model selection.

Experiments with Horovod Hybrid. Our experiment with the Horovod Hybrid gave an anti-climactic result: the intrinsic network overheads of Horovod meant the hybrid is often slower than regular CEREBRO with some workers being idle! We realized that mitigating this

Table 3.6. Parameter grid used to randomly sample configuration for Section 3.6.3.

	Values sampled from
Model	[ResNet18, ResNet34]
Learning rate	$[10^{-5}, \ldots, 10^{-1}]$
Weight decay coefficient	$[10^{-5}, \ldots, 10^{-1}]$
Batch size	$[16, \ldots, 256]$

issue requires more careful data repartitioning. We deemed this complexity as perhaps not worth it. Instead, we propose a simpler resolution: if $|S|$ falls below p but above $p/2$, use CEREBRO; if $|S|$ falls below $p/2$, just switch to Horovod. This switch incurs no extra overhead. Due to space constraints, we skip the details here and explain this experiment further in Appendix A.

3.6.3 Experiments with AutoML Procedures

We experiment with two popular AutoML procedures: HyperOpt [41] and ASHA [175]. For HyperOpt, we compare CEREBRO and Spark as the execution backends. Spark is a backend supported natively by HyperOpt; it distributes only the models, i.e., it is task-parallel on fully replicated data. For ASHA, we compare CEREBRO and Celery as the execution backends. We use *ImageNet*, GPU cluster, and PyTorch. Training configs are sampled from the grid shown in Table 3.6. For CEREBRO data is partitioned without replication; for Spark and Celery the dataset is fully replicated.

Both HyperOpt and ASHA keep exploring different configs until a resource limit is reached. For HyperOpt, this limit is the maximum number of configs; for ASHA, it is the maximum wall-clock time. During the exploration HyperOpt uses Bayesian sampling to generate new configs; ASHA uses random sampling. For both methods, the generated configs are dependent on the completion order of configs across task-parallel workers. Thus, it is impossible for CEREBRO to *exactly* replicate HyperOpt or ASHA ran with task-parallelism. However, we can closely *emulate* HyperOpt and ASHA on CEREBRO by making the number of simultaneously trained configs ($|S|$) equal to the number of workers (p) and without making any changes to CEREBRO.

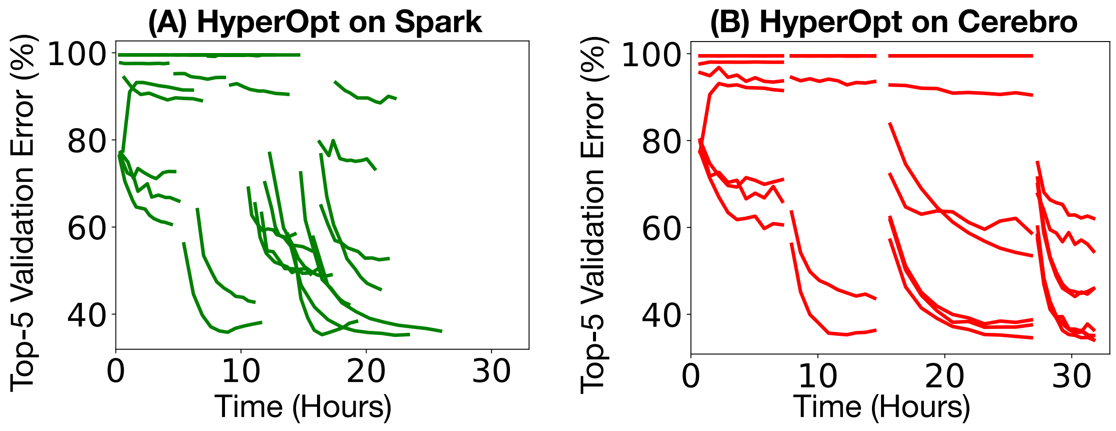


Figure 3.12. HyperOpt learning curves by time.

HyperOpt. We run an experiment using HyperOpt with a max config budget of 32. We train each config for 10 epochs. With this configuration, HyperOpt on CEREBRO (resp. Spark) took 31.8 (resp. 25.9) hours. Figure 3.12 shows all learning curves. We found that the slightly higher (23%) runtime of CEREBRO is mainly due to the lower degree of parallelism ($|S| = 8$). However, this issue can be mitigated by increasing the number of simultaneously trained configs. Although individual configs are not comparable across the two systems, the best errors achieved are close (34.1% on CEREBRO; 33.2% on Celery).

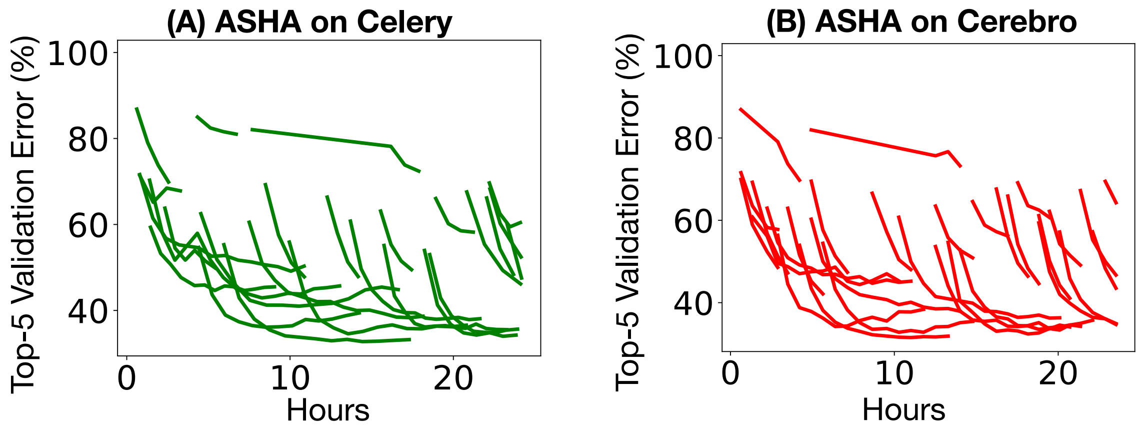


Figure 3.13. ASHA learning curves by time.

ASHA. We use ASHA with a max epoch budget (R) of 9, a selection fraction (η) of 3, and a time limit of 24hr. With these settings, ASHA trains for a maximum of 13 epochs over 3 stages: 1, 3, and 9 epochs. Only the more promising configurations are trained for more epochs. In the given time limit, ASHA on CEREBRO (resp. Celery) explored 83 (resp. 67) configs. Figure 3.13 shows all learning curves. Like HyperOpt, even though the configs are not directly comparable, the best errors achieved are close (31.9% on CEREBRO; 33.2% on Celery). More details about this experiment and experiments with another AutoML procedure (HyperBand) are presented in Appendix A.

3.7 Discussion and Limitations

Applications. CEREBRO is in active use for time series analytics for our public health collaborators. In the case study from Section 3.1, CEREBRO helped us pick 16 deep net configs to compare. To predict sitting vs. not-sitting, these configs had accuracies between 62% and 93%, underscoring the importance of rigorous model selection. The best configs gave a large lift of 10% over their prior RandomForest model based on hand-engineered time series features. We plan to use CEREBRO for more domain science applications in the future on time series, video, graph, and text data.

Open Source Systems. CEREBRO is open sourced and available for download [1]. MOP's generality also enabled us to emulate it on existing data-parallel systems. Pivotal/VMware collaborated with us to integrate MOP into Greenplum by extending the MADlib library [119] for running TensorFlow on Greenplum-resident data [197, 278]. Greenplum's customers are interested in this for enterprise ML use cases, including language processing, image recognition, and fraud detection. We have also integrated CEREBRO into Apache Spark [77]. CEREBRO-Spark can run MOP on existing resource managers such as YARN and Mesos. Alternatively, one can also deploy CEREBRO as a standalone application by wrapping it as tasks accepted by the resource manager. We leave such an extension to future work.

Other ML Model Families. We focused primarily on deep nets due to their growing popularity, high sensitivity to model configurations, and resource intensiveness. However, note that MOP and CEREBRO's ideas are directly usable for model selection of *any* ML models trainable with SGD. Examples include linear/logistic regression, some support vector machines, low-rank matrix factorization, and conditional random fields. In fact, since linear/logistic regression can be trivially expressed in the deep learning tools's APIs, CEREBRO will work out of the box for them. CEREBRO's high memory efficiency makes it easier for users to store the entire large datasets in distributed memory, which can significantly reduce runtimes of such I/O-bound ML models.

3.8 Conclusion

The high costs associated with model selection are one of the major obstacles to the broader adoption of modern DL/AI. To mitigate this issue, we present a simple but novel and highly general form of parallel SGD execution for model selection workloads, MOP, that raises the resource efficiency of deep net model selection without sacrificing accuracy or reproducibility. MOP is also simple to implement, which we demonstrate by building CEREBRO, a fault-tolerant deep net model selection system that supports multiple popular deep learning tools and model selection procedures. Experiments with large ML benchmark datasets confirm the benefits of CEREBRO.

Chapter 3 contains material from "Cerebro: A Data System for Optimized Deep Learning Model Selection" by Supun Nakandala, Yuhao Zhang, and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 13, Issue 12, July 2020. The dissertation author's contribution was in the conceptualization of the system, parts of the implementation, and parts of the experiments. The code for our system is open source and is available on GitHub: https://github.com/ADALabUCSD/cerebro-system.

Chapter 4

CEREBRO on Data Systems: Bridging the Gap Between Data Systems and DL Workloads

4.1 Introduction

We now move our eye to the practicality concerns of DL, and in this chapter, we will discuss the challenges and solutions of bringing DL to existing data systems and bridging the gap between in-database enterprise data and DL workloads. The DBMS community has long worked on bringing ML closer to the home of business-critical datasets in enterprises: DBMSs and other data systems. This paradigm of “In-DBMS ML” (or “In-data system ML”) has waxed and waned over the last 20 years, with 3 general waves of work. It now merits a revisit in the era of DL.

4.1.1 Lessons from In-RDBMS ML

In the first wave of in-RDBMS ML, DB vendors built “data mining tools” that scaled a few ML algorithms to DB-resident data [64, 15, 224]. They enabled access to ML from the SQL console. But as ML algorithms grew in complexity, a second wave of unified implementation abstractions were devised for in-data system ML [91, 70]; MADlib [120] and Spark MLlib [200] are key examples. The third wave is seeing cloud DBMS vendors adding more in-RDBMS ML support, e.g., Google’s BigQuery ML [9, 14, 8], as well as invoking DL from DBMSs [10, 19].

In this context, DBMS and cloud vendors are increasingly asking: *"How to enable seamless support for DL over DB-resident data?"*. The past waves of in-RDBMS ML offer at least four lessons.

1. The main user base of in-RDBMS ML tools are not Python-oriented data scientists but SQL-oriented business analysts. Such users increasingly want access to DL training and inference *from within the SQL console*. As per estimates by the MADlib team [2], about 20-25% of Greenplum customers today use its in-RDBMS ML analytics capabilities alongside SQL analytics.

2. Although *governance and provenance* were always important for enterprises in sensitive domains such as financial and health care, they now have renewed urgency for *all companies* including the tech giants, due to new laws such as GDPR [72] and CCPA [221]. Companies will likely start frowning upon DL users manually exporting, copying, and moving business-critical data around in an ad hoc manner. Although one could program to automate such processes, and use services like MLFlow and Kubeflow [204, 156] for governance and provenance tools, it is still an extra burden for the enterprise users to learn, especially when they are already familiar with established DBMS support for governance/provenance.

3. It is far too tedious for DBMS developers to reimplement DL algorithms. So, one must *preserve the usability of DL tools* such as TensorFlow for specifying complex DL workloads. This also allows analysts to just reuse DL training specification programs written by data scientists or others.

4. Parallel RDBMSs already offer a mature execution engine on sharded large-scale data. But state-of-the-art distributed DL execution tools such as Horovod [257] are still notoriously painful to set up, operate, and debug [24]. This presents parallel RDBMSs/data systems an opportunity to *bridge the gap on scalable execution*.

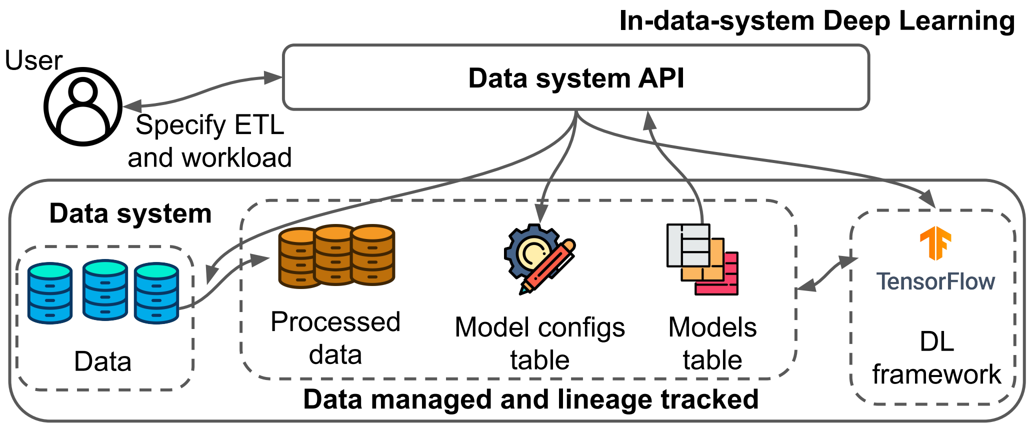


Figure 4.1. In-data-system DL. Data system invokes DL tool and helps mitigate data provenance/governance issues.

Overall, we see two contrasting paradigms for how DL is brought to DB-resident data. The DL user can export the data to a file system, invoke a DL tool manually, and manage all derived data/metadata/artifacts on their own. Alternatively, in the "in-data-system DL" approach, ETL and the DL workload are orchestrated by a data system, as Figure 4.1 illustrates. Crucially, this approach leaves room for *implementation flexibility* on how exactly the DL tool consumes data; this flexibility opens up possibilities that we will explore later.

4.1.2 Toward In-Data System DL

Apache MADlib has recently pioneered in-DBMS DL support [12]. DL workload is specified using Keras APIs, enabling business analysts to reuse DL configurations written by, say, data scientists. MADlib ships mini-batch data from the DB to a TensorFlow function invoked in a DBMS User Defined Function (UDF)/User Defined Aggregate Function (UDAF). For distributed execution, MADlib used the "model averaging" (MA)[1] heuristic for SGD [327, 91]. Alas, MA has poor convergence behavior for highly non-convex DL [214]. Thus, this approach is sub-optimal for bringing DL to DBs.

We observe that MA misses a major opportunity for parallelism in DL: *model selection*. ML theory teaches us that tuning hyperparameters is crucial, and this requires training many

[1]In addition to MA, MADlib has adopted one of the approaches we will evaluate [13].

models [258, 160]. Often, DL users also compare alternate neural architectures, alter the base features, etc. Thus, model selection in practice often leads to dozens, if not hundreds, of models to train in one go [86, 214].

Exploiting the above observation, recent work proposed a new approach to distributed DL model selection called Model Hopper Parallelism (MOP) [214, 213, 161]. MOP is a *hybrid* of sharded data parallelism and task parallelism. MOP works as follows: train different models on different workers in parallel for one *sub-epoch* on their local shards, checkpoint and "hop" the models across workers, and restart training the *same epoch* on the next worker's shard. MOP is a form of bulk *asynchronous* parallelism since it imposes no barrier synchronization across workers, unlike Bulk Synchronous Parallel (BSP) data systems. Overall, MOP was shown to be the most resource-efficient approach to distributed DL model selection [214].

4.1.3 Focus of this Chapter

Given the benefits of MOP we ask: *"How to bring MOP-based DL to DB-resident data?"* We find that there is no single "best" approach, and there is an interesting tradeoff space of alternative approaches. This paper explains these approaches, contrasts them analytically, and compares them empirically with large-scale DL workloads. We use Greenplum as the archetype but emphasize that the approaches compared are generic and applicable to any parallel RDBMS. Thus, *our results could be of wide interest to all DBMS and cloud vendors*.

We seek approaches that *do not change the code* of the data system. This eases practical adoption but restricts how MOP can be applied. For instance, Spark now supports flexible scheduling of workers [76]; this made it easy to integrate MOP with Spark in the Cerebro system [1]. But parallel RDBMSs such as Greenplum, AWS Redshift, etc., use BSP across workers, conflicting with MOP's asynchrony. We have *multiple axes of comparative evaluation*, including *runtime efficiency*, *ease of governance*, *implementation difficulty*, and *portability*. Section 4.4 explains all approaches and Section 4.5 compares them in detail, but as a preview, Figure 4.2 shows the approaches on the first two axes.

We compare 4 new approaches: (1) Fully in-DBMS MOP using UDAF, which has been adopted by MADlib [13] (2) Partially in-DBMS MOP using Concurrent Targeted Queries (CTQ) (to be introduced in Section 4.4.2); (3) In-DB but *not* in-DBMS (data is in DB but all operations are not) MOP with Direct Access (DA); and (4) Regular out-of-DBMS approach using Cerebro-Spark. MA is largely dominated by the UDAF approach but all the other approaches fall on the Pareto frontier. For instance, the out-of-DBMS Cerebro-Spark approach and in-DB DA approaches are much more efficient than UDAF but may be harder to govern in a production environment. The CTQ approach offers a middle ground on these two axes.

Our comparative analyses of these approaches expose more interesting gaps. For instance, with theoretical and simulation analyses, we show that the efficiency gap between CTQ and UDAF grows wider when the models and hardware are more heterogeneous, even up to 6x in a realistic scenario. Finally, an extensive empirical comparison using the ML benchmark datasets ImageNet and Criteo shows that the real runtime gaps between UDAF and DA be as high as 3x. Overall, our experiments and analyses show that it is beneficial to bring MOP-based DL to DB-resident data, but it is non-trivial to meet all practical desiderata. We hope our results spur more conversations in the DB and cloud industries on how best to support DL on DB-resident data.

In summary, we make the following contributions:

- To the best of our knowledge, this is the first paper to analyze the tradeoffs and design alternatives of supporting large-scale DL model selection on DB-resident data.
- We show a spectrum of possible approaches on the Pareto frontier of efficiency, ease of governance, and other practical desiderata. In particular, we show a new approach that is in-DB but not in-DBMS, posing new accessibility questions for DB vendors.
- We perform a formal analysis of the limits of the efficiency gaps between the new approaches.

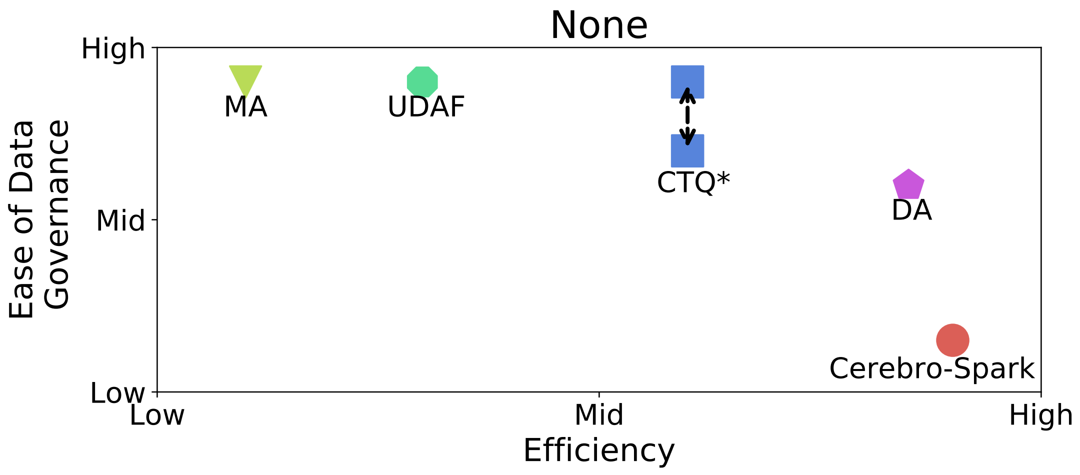


Figure 4.2. Tradeoffs of ease of data governance vs. efficiency for various approaches. *Depending on an implementation detail CTQ may have the same ease of governance as MA and UDAF; see Section 4.5.2 for details.

- We perform an extensive empirical comparison of the approaches using large ML benchmark datasets to evaluate their runtimes, scalability, and internal design tradeoffs.

4.2 Constraints and Challenges in Bringing DL to DBMSs

We first consider in-DBMS DL that relies only on UDAFs/UDFs without modifying the internal code of the DBMS. We also use the data handling functionalities of the DBMS. It is challenging to implement because of constraints that many parallel DBMSs share. We summarize these constraints as follows:

Bulk synchronous parallelism (BSP). Each query executes in an all-or-nothing manner on a dataset that is sharded across workers. A synchronization barrier is injected at the end of every query. There is no trivial way at the SQL/UDF level to poll partial results.

textbfNo message-passing among workers. Some of the existing DL systems rely on protocols such as MPI or RPC for communication, but to enable these functionalities at the UDF level would require modifications to the DBMS and/or substantial efforts. Hence the preferable communication method among workers is the pipes provided by the DBMS. This constraint would make some distributed DL paradigms especially hard to implement.

One query at a time. For each database connection session, only one query is permitted at any time. Using multiple clients and DB sessions for the same query is a way to achieve parallelism by manually dissecting the query into subqueries and unifying the results on the client-side, it may be considered as an anti-pattern as now the query planning takes place out-of-DBMS.

Data access through DBMS. In a DBMS, data is usually compressed and stored on disk as pagefiles (physical files on disk that contain database pages.). To access data, one must go through the DBMS query stack. If the data is frequently and iteratively accessed as we see in DL training, such repeated accessing and decompression could bring serious overheads.

4.3 The Fitness of Prior Art for In-DBMS DL

There has been a lot of work on distributed DL training. For a detailed background, see Section 3.2. However, most of the techniques do not have or assume a trivial data layer. Adjustments must be made to integrate them into an existing data system. The DBMS has constraints that render many of the approaches unsuitable or difficult to implement. We translate the constraints of Section 4.2 into the following requirements for distributed DL paradigms for amenability to the in-DBMS setting.:

- **Centralized communication.** As mentioned in Section 4.2, we want the communication pattern to be as simple as possible. P2P communication is typically not allowed.

- **Coarse-grained parallelization.** The training would better be parallelized at epoch instead of mini-batch level. Since we will embed the training jobs as data system tasks/queries, fine-grained parallelization will lead to massive number of queries that can cause heavy overheads.

- **Data-parallelism.** The data is already partitioned in the data system. Fully replicating the entire data across workers is not desirable and may not even be feasible at large scales.

Table 4.1. Summary of various parallel paradigms' fitness for in-data-system DL.

	Centralized communication	Coarse grained	Data-parallel	Fast convergence
Task Parallel	✓	✓	✗	✓
Model Avg.	✓	✓	✓	✗
Param. Server	✓	✗	✓	✓
Horovod	✗	✗	✓	✓
MOP	✓	✓	✓	✓

- **Fast convergence.** In order to save computational and resource costs of model selection, we want the models to converge fast in terms of number of epochs, ideally resembling the learning curves obtained by the gold-standard sequential SGD.

Next we explain the major distributed DL model selection approaches in the literature and explain how well they fit (or not) the above constraints. Table 4.1 summarizes our comparative analysis.

Task Parallel. In this paradigm, different model configs of the model selection workload run on different workers in a task-parallel manner. Example tools include Python Dask, Celery, Vizier [104], and Ray [206]. Workers locally run sequential SGD on the whole dataset. Thus, this approach provides the best convergence efficiency. There is no communication across workers during training. Still, it *requires full data replication on each worker*, which is inefficient, and may not even be feasible for large sharded datasets in DBMSs.

Model Averaging (MA). In BSP systems such as TensorFlow with model averaging [17], data is sharded. The model configs are trained in parallel one-by-one. Every model is broadcasted and trained on each worker's data shard independently. Then a merge step takes place on the master; it averages the weights (or gradients). This approach is a potential candidate and has been adopted by MADlib [12]. Alas, it converges poorly for DL models, which are highly non-convex [263]. Nevertheless, since it satisfies most of the constraints, we include it as a key baseline in our experiments.

Fine-grained Parallel. These paradigms are similar to BSP, but they work at a finer granularity at the mini-batch level. The communication pattern can be centralized or decentralized.

The most prominent example of centralized paradigms is Parameter Server (PS) [177]. The best example for decentralized paradigms is Horovod [257]; it adopts HPC-style techniques to enable synchronous all-reduce SGD. These methods all have good convergence behavior but very high communication costs. They too are not good candidates because of the granularity. Horovod further requires P2P communication patterns that are not allowed in most data systems.

Model Hopper Parallelism (MOP). MOP used in system Cerebro [214] is recent progress towards resource-efficient DL. This is a hybrid of task- and data-parallelism. Each worker is assigned one model config from the model selection workload and trains the model with its local data shard; this process is called one sub-epoch. When one sub-epoch finishes, the model is passed to other data shards for further training. After several sub-epochs, every model finally has seen the entire dataset, and that is one epoch of training. Overall, a model *hops* from one worker to another in-between sub-epochs. The scheduling is done via an asynchronous random scheduler that works well on heterogeneous workloads and supports fault tolerance. MOP fits all our requirements because communication-wise, it has a centralized pattern and low cost, for it works at sub-epoch granularity. Data-wise, it works nicely with sharded data. Finally, it offers equivalency to sequential SGD, which has the highest convergence efficiency. Hence, we decided MOP would be a better choice for in-DBMS DL.

4.4 Overview of CEREBRO on Data Systems

Given the benefits of MOP, the question becomes how to bring MOP-based DL to DBMS-resident data. There are multiple possible approaches due to the implementation flexibility. To better explain these alternatives, we first divide the components of MOP execution into five layers of design decisions: Interface, Scheduling, Execution, Data Access, and Storage. Each layer can be implemented in flexible ways. Figure 4.3 summarizes the architectural alternatives.

figures/

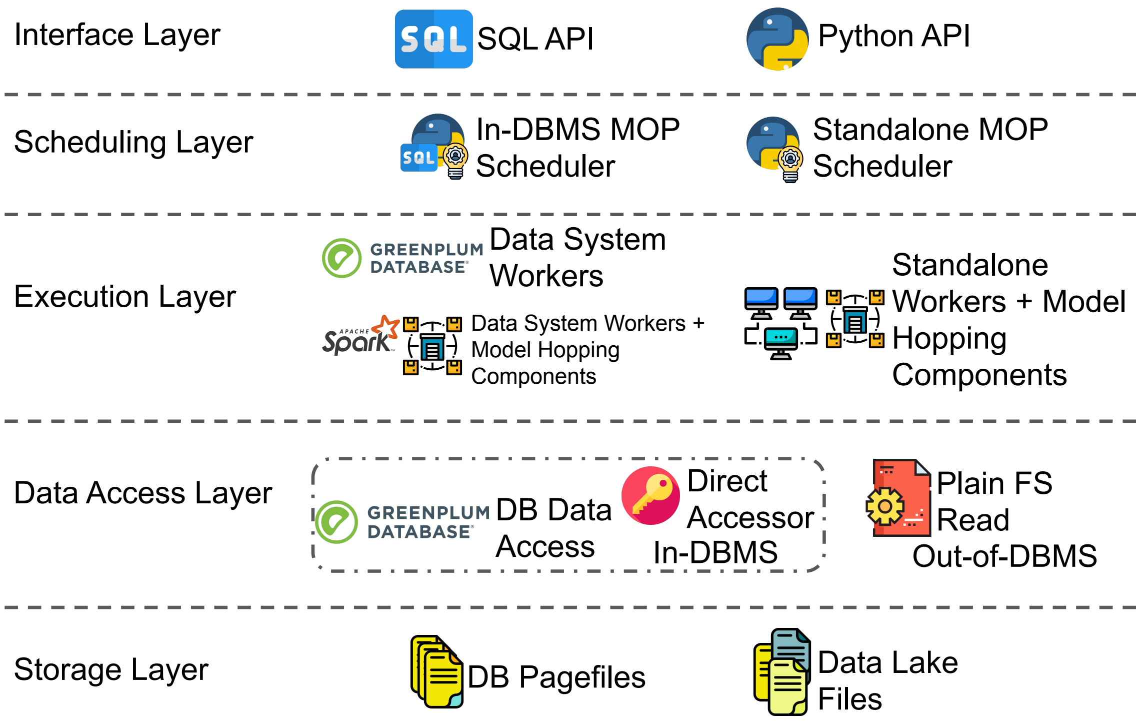


Figure 4.3. Design alternatives for MOP in DBMS.

1. **Interface layer**: the high-level APIs that take in the user's DL model selection workload; it could be implemented in SQL, familiar to business analysts, or in Python, familiar to data scientists.

2. **Scheduling layer**: the scheduler orchestrates and manages placements of training units. We could implement the scheduler as an in-DBMS procedure or use a standalone MOP scheduler.

3. **Execution layer**: execution engine invokes DL tools and conducts model training via mini-batch SGD. We could use the data systems' execution engine or resort to standalone Cerebro for this layer. Model hopping can be cast as SQL queries or be implemented as separate components with other communication methods.

4. **Data Access and Storage layer**: we could leave the data in DBMS or export them. There are also multiple ways to access the data; if data is in a data lake, access is trivial.

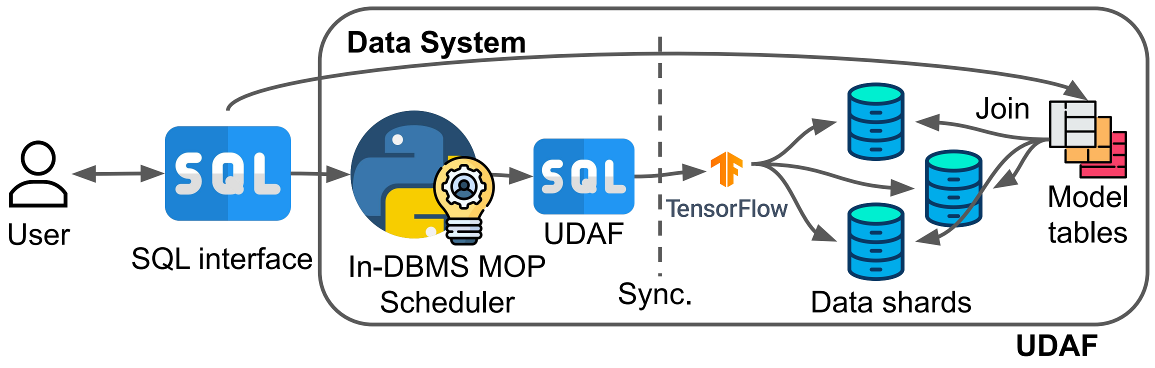


Figure 4.4. UDAF approach. Fully in-DBMS.

Otherwise, if the data is in DBMS, we can rely on DBMS's native data accessor or use a technique we call Direct Access to bypass the whole query stack and access the data from its physical storage directly.

Because of these flexibilities, there are various approaches for the end-to-end implementation of MOP. We find four interesting canonical approaches: (1) Fully in-DBMS MOP using User-Defined Aggregate Functions (UDAF); (2) Partially in-DBMS MOP using Concurrent Targeted Queries (CTQ); (3) In-DB but not in-DBMS MOP using Direct Access (DA); and (4) Regular out-of-data-system approach using Cerebro-Spark. We use Greenplum as the archetype but emphasize that the approaches compared are generic and applicable to any parallel DBMS. We do not claim these are the only possible approaches; rather, we find these are prototypical examples of feasible approaches based on the combinations of design decisions in Figure 4.3. We now introduce each approach and dive into the analysis and comparisons later in Section 4.5. We summarize the design choices for each approach in Table 4.2.

4.4.1 User-Defined Aggregate Functions (UDAF)

This approach implements MOP as DBMS extensions with UDFs and UDAFs. On the 5 design decisions of this approach and other approaches to be introduced, please refer to Table 4.2. UDAF approach is called fully in-DBMS because all functions are in-DBMS procedures, and both data and models are stored in DBMS.

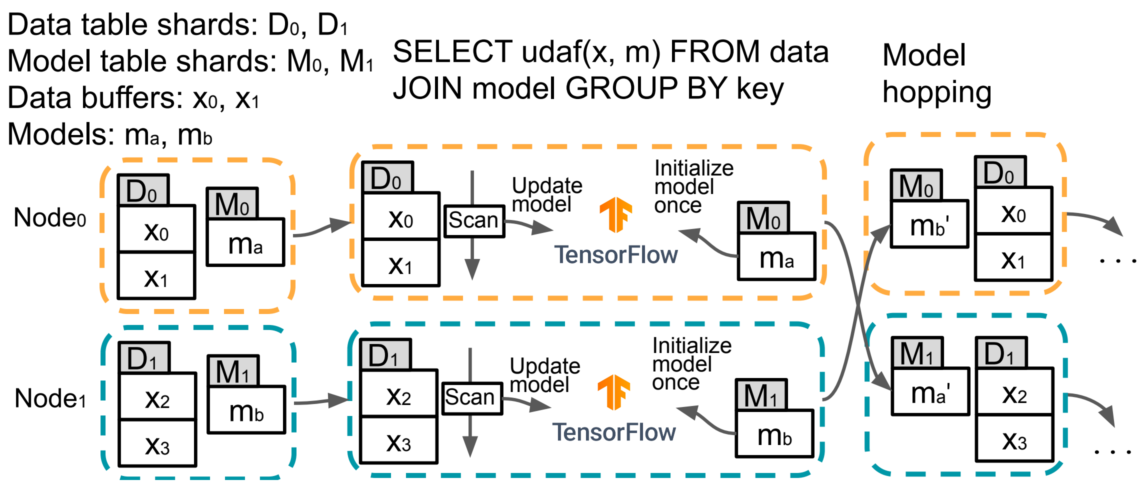


Figure 4.5. Conceptual illustration of one sub-epoch of UDAF.

Table 4.2. Summary of each approach's design for the 5 layers. IN: in-DBMS. OUT: out-of-DBMS.

Approach	Interface	Scheduling	Execution	Data Access	Storage
UDAF	SQL	IN	IN	IN	IN
CTQ	Python	OUT	IN	IN	IN
DA	Python	OUT	OUT	OUT	IN/OUT
Cerebro-Spark	Python	OUT	OUT	OUT	OUT

Figure 4.4 illustrates the approach. It maintains a data table and a model table storing the DL model selection workload, with each row containing a model. Both tables are sharded based on distribution keys. These keys are used by the DBMS to determine where the rows are stored. The rows are distributed across worker nodes by the master node matching the values of a designated distribution key column. So all the rows with the same distribution key end up in the same worker node. By manipulating these keys, we can control the affinity of data/model. The user first defines the model architectures and workloads through a SQL interface. DBMS then invokes the MOP scheduler implemented in UDF.

This approach's scheduler is synchronous due to the BSP nature of in-DBMS execution, in contrast to the asynchronous random scheduler that standalone MOP adopts. It uses a simple round-robin heuristic for placing models on data shards. The scheduler translates the workload into UDAF queries dispatched and executed on the joined table of data and model. These UDAFs

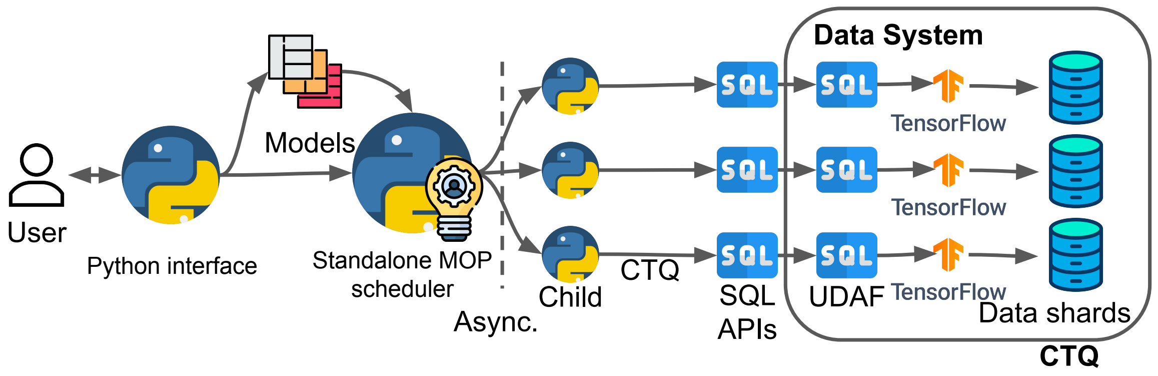


Figure 4.6. CTQ approach. Partially in-DBMS.

subsequently invoke DL tool (we use TensorFlow/Keras) for training. It schedules a batch of sub-epochs on the workers at a time and waits for completion. After several batches, one epoch is completed; it then repeats the process to train for multiple epochs.

Conceptually, each UDAF is a query of `SELECT udaf(...) FROM data JOIN model GROUP BY key`. Figure 4.5 illustrates the execution of one sub-epoch batch. Data is pre-packed into buffers and stored in a sharded table. Models are stored similarly in another sharded table. On each physical node of the DBMS, there are multiple rows of data buffers and only one row of model. During the execution, the model row is fed to DL tools to initialize the model, which will be stored as the aggregation state. The worker then scans the data shard and feeds each data buffer to the DL tool, which unpacks the buffers and generates mini-batches for training and updating the stored model. After scanning is done, the scheduler redistributes updated models to different physical nodes by manipulating their distribution keys. The data table, on the other hand, never redistributes.

4.4.2 Concurrent Targeted Queries (CTQ)

This approach is built upon CTQs, a DBMS feature we will explain shortly. In contrast to UDAF, it has a Python interface, uses a standalone MOP scheduler and out-of-DBMS model hopping components. We chose to store and hop models out of the DBMS for implementation simplicity; it is technically possible to keep models governed by DBMS, which can raise this

Table 4.3. Conceptual comparison of various architectural approaches of integrating MOP with DBMS. *CTQ can have highest or high ease of governance, depending on whether models are governed by DBMS. †If node RAM is insufficient, swap is needed for DA and the blowup could rise up to 2x.

	Efficiency	Governance	Storage Blowup	Implementation Difficulty	Portability	Design Anti-patterns
UDAF	Medium	Highest	None	Medium	Medium	No
CTQ	High	High-Highest*	None	Medium	Medium	Yes
DA	Highest	High	None - 2x†	Hard	Low	Yes
Cerebro-Spark	Highest	Low	2x	Easy	High	N/A

approach's ease of governance (see Section 4.5.2). Since some core computations run outside the DBMS, we call this approach partially in-DBMS.

We now explain what a CTQ is. In a parallel DBMS, tables are sharded according to distribution keys. When a query only affects one shard, e.g., with a predicate that filters on the distribution key, the query processor will dispatch a query plan to that specific shard only. Such feature is sometimes called targeted query and commonly available [6, 202, 28]. Meanwhile, most DBMSs also allow concurrent queries. Therefore, we can assume more fine-grained control over the execution by issuing targeted queries concurrently. We name this trick Concurrent Targeted Queries (CTQ).

Figure 4.6 shows the CTQ approach. The user interacts with a Python interface to define models and workloads. It then invokes a standalone MOP scheduler, as described in [214]. This scheduler works differently from the one used in the UDAF approach; it orchestrates DL training by spawning children processes that contain DBMS connections and using them to issue CTQs. Meanwhile, models are hopped outside the DBMS; we use a shared filesystem for this task. Conceptually, each CTQ is a query of `SELECT udf(model) FROM data WHERE key=x`. Each DBMS node loads the assigned model from the shared filesystem and uses its local data shard to train the model, then checkpoints the updated model back to the filesystem. This concludes one sub-epoch; after every model has visited every data shard once, it is called one epoch.

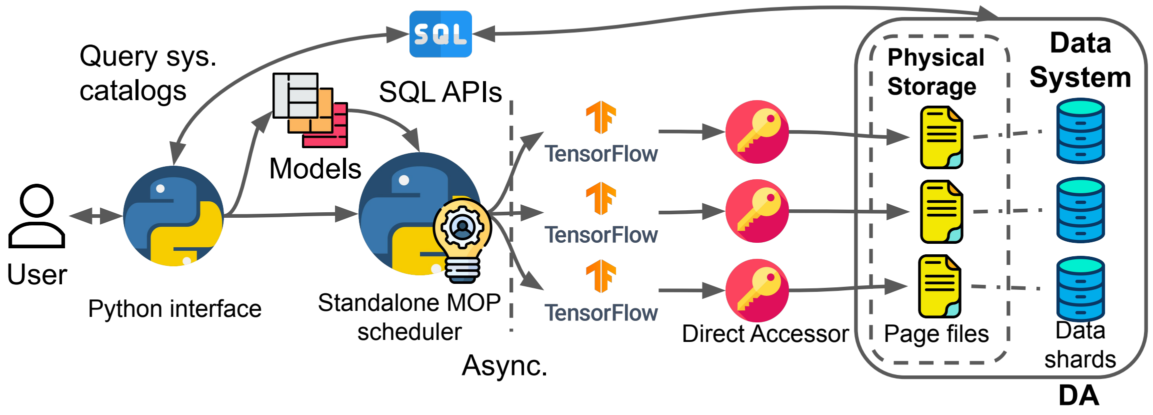


Figure 4.7. DA approach. In-DB but not in-DBMS.

4.4.3 Direct Access (DA)

This approach further deviates from UDAF and CTQ by employing a method we call Direct Access, bypassing the entire query processor of DBMS and accessing the on-disk pagefiles directly. This way, there is enough freedom to plug in and run standalone Cerebro [214] system but without exporting the data. This approach is called in-DB but not in-DBMS because although the DBMS still governs data, all executions are out of DBMS.

Figure 4.7 illustrates DA. The user talks to a Python interface to define workloads and query necessary system catalogs. DA then uses the standalone Cerebro for scheduling and execution. Workers perform training on the data table's sharded pagefiles directly through DAs. DAs first retrieve the pagefiles' location, mapping, layout, and compression information from system catalogs. Then they emulate DBMS's access methods to fetch the pages' contents and feed the data to Cerebro. The latter then consumes the data and runs MOP to train the workload. Notably, this approach is very generalizable and not limited to MOP execution; one can essentially plug in any data-parallel training frameworks like Pytorch or Horovod. To demonstrate the generality, in addition to MOP, we will also implement a fine-grained data-parallel approach with DA using Pytorch DDP. We will show the evaluations in Section 4.6.1.

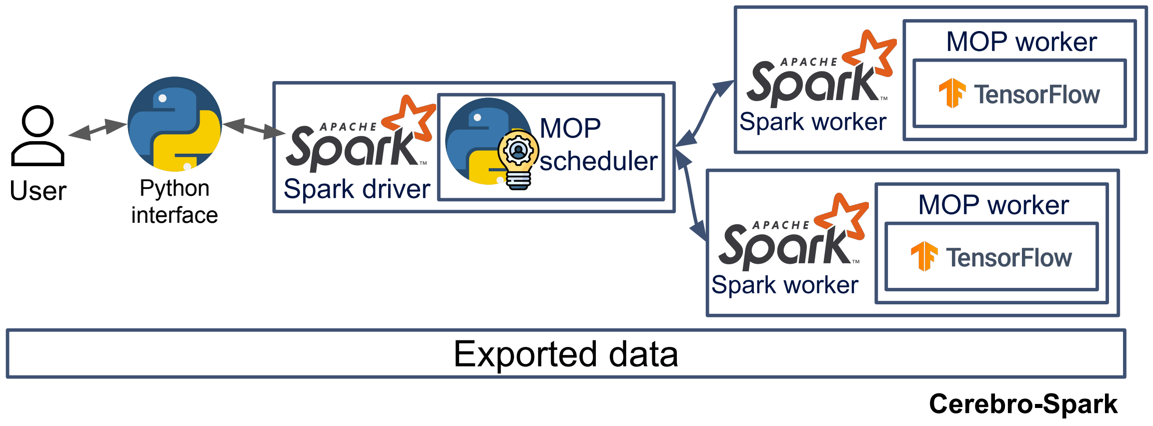


Figure 4.8. Cerebro-Spark approach. Fully out of DBMS.

4.4.4 Cerebro-Spark

Cerebro-Spark is a regular out-of-DBMS approach that exports data to filesystem, runs ETL processes, and feeds data to the DL tools. It uses the data system (Spark) workers and stores data as plain files. The DBMS does not participate in the training and loses the governance of data.

Figure 4.8 illustrates the architecture. The user defines workloads through Python APIs. The standalone MOP scheduler initializes MOP workers by embedding them as long-running Spark tasks. It then communicates with these workers and orchestrates the training just like in the standalone Cerebro system. In addition to Cerebro-Spark, we will also evaluate other frameworks such as Pytorch DDP and Hyperopt-Spark in Section 4.6.

4.5 Comparative Analyses of Approaches

With all the approaches introduced, we now compare and analyze them on 6 major axes: runtime efficiency, ease of governance, storage blowup (defined as the actual storage usage divided by the original data size), implementation difficulty, portability, and design anti-patterns. Table 4.3 shows a conceptual comparison. These axes represent the desiderata and we find that no single approach can fulfill all of them. The more the approach is in-DBMS, the lower the runtime efficiency but the higher the ease of governance and vice versa. We will discuss the reasons in Section 4.5.1 and 4.5.2. In terms of storage, Cerebro-Spark has 2x blowup because of exporting. The situation is more complicated on the implementation difficulty and portability axes, and we will give a more rigorous analysis in Section 4.5.3 and 4.5.4. As for the last axis, CTQ and DA both introduce design anti-patterns since they are not fully in-DBMS: CTQ introduces a user-level anti-pattern by issuing multiple queries concurrently from outside of the SQL console, while MADlib and many other tools makes a single query inside a SQL console. This violates *One query at a time* mentioned in Section 4.2. DA has anti-patterns that violate *No message-passing among workers* and *data access through DBMS* from Section 4.2. In the rest of this section, we pick 4 most interesting axes and analyze them in more detail.

4.5.1 Runtime Efficiency

This is one of the most important desiderata. Several factors affect runtime: DL tool invocation, data access, model hopping, and schedule makespans (end-to-end runtime of the generated schedule).

- **DL tool invocation.** Both UDAF and CTQ invoke the DL tools through wrappers, whereas DA and Cerebro-Spark do not. Such wrappers may be a source of inefficiency.
- **Data access.** Both UDAF and CTQ access data through DBMS. They could be bottlenecked by data transmission[2], especially when the data is compressed and tweaked by the DBMS, e.g., TOAST-ed [20]. DA can mitigate this issue; by accessing the physical pagefiles directly and caching data, it provides similar efficiency to Cerebro-Spark, which also reads from filesystem and caches data in memory.
- **Model hopping.** Model hopping might be another source of inefficiency. CTQ, DA, and Cerebro-Spark all do model hopping outside of the DBMS and have similarly low overheads on this end, as [214] pointed out. On the other hand, the UDAF approach relies on the DBMS to hop models through `JOIN` between the data and model tables. This

[2]Active development by the MADlib team is going on to mitigate this issue.

Table 4.4. Notation for discussion on scheduling makespans.

Notation	Description
$\mathbb{M}, M$	Set of models and the cardinality of it
$\mathbb{W}, W$	Set of workers and the cardinality of it
$\mathbb{L}$	Set of each model's per sub-epoch runtime
$\mathbb{L}_i$	For UDAF only. Batch of models scheduled for the i-th sub-epoch
m_x	The x-th model
l_x	The per sub-epoch runtime of the x-th model
l_s	A scale representing the runtimes of fast models
l_m	A scale representing the runtimes of slow models
p	Probability of a model being a fast model
T_u, T_c	End-to-end runtimes for sync. and async. MOP, respectively
η	Theoretical upper bound of the speedup T_u/T_c

`JOIN` may bring some overheads, especially if the models are large. In later experiments (Section 4.6.2), we will indeed see UDAF is much slower than CTQ on model hopping. However, even for UDAF, model hopping still incurs negligible runtime compared to other components.

- **Scheduling makespans.** CTQ, DA, and Cerebro-Spark all employ the same asynchronous random scheduler. This scheduler's robustness on heterogeneous workloads/workers has been tested in [214]. However, UDAF uses a synchronous round-robin scheduler, which may not work very well with heterogeneity. We show visualizations of potential scenarios in B. How large is the gap between these two schedulers, and could it be a major performance bottleneck? We now analyze the differences theoretically between sync. and async. MOP and later verify it empirically in Section 4.6.2. Table 4.4 presents all notations used in this section.

Let there be a set of model configs $\mathbb{M}$ and a set of workers $\mathbb{W}$. $|\mathbb{M}| = M$, and $|\mathbb{W}| = W$. Assume the workers to be identical and each worker contains the same amount of data. Let l_x denote the per sub-epoch runtime of model config m_x and $\mathbb{L} = \{l_x\}$. For analysis simplicity, let $\mathbb{L}$ be a two-mode right-tailed distribution, i.e., most models are fast and have per sub-epoch runtime of l_s, while only some are slow and take l_m, $l_m \gg l_s$. Let p be the probability of l_x being fast: $p = Pr(l_x \sim l_s)$. We now analytically compute the per-epoch makespan T_u and T_c for sync.

Table 4.5. Workloads.*architectures similar to VGG16 and ResNet50, respectively.

Dataset	Model arch.	Batch size	Learning rate	Regularization	Epochs
ImageNet	{VGG16*, ResNet50*}	{32, 256}	$\{10^{-4}, 10^{-6}\}$	$\{10^{-4}, 10^{-6}\}$	10
Criteo	3-layer NN, 1000+500 hidden units	{32, 64, 256, 512}	$\{10^{-3}, 10^{-4}\}$	$\{10^{-4}, 10^{-5}\}$	5

and async. MOP, respectively. We have the following two propositions, the proofs to them can be found in B.

Proposition 1. Speedup of async. over sync. MOP is:

$$\frac{T_u}{T_c} = p^W \frac{l_s}{\bar{l}} + (1 - p^W)\frac{l_m}{\bar{l}}. \tag{4.1}$$

Proposition 2. Theoretical upper bound of the speedup is:

$$\eta = \frac{l_m}{pl_s + (1-p)l_m}. \tag{4.2}$$

Section 4.6.2 shows an experiment that verifies the analysis.

4.5.2 Ease of Governance

As we mentioned earlier, data governance/provenance now has renewed urgency for all enterprises and even the Web companies, because of the new regulations and laws like GDPR [72] and CCPA [221]. Among the four approaches, UDAF provides the best support for governance/provenance, as it keeps both the dataset and the models in DBMS, which already has built-in governance support. CTQ and DA both use DBMS to govern data. For CTQ, we chose to store models out of DBMS for simplicity, but it is technically possible to keep models in DBMS; this way, it can provide similar ease of governance as UDAF. DA, which relies on external Cerebro, does not manage models with DBMS and thus, loses some ease of governance. Cerebro-Spark does not come with existing governance support and may impose other security issues due to the ad hoc data export and copying. To regain governance, one has to maintain exporting scripts and seek help from external services like MLflow or Kubeflow [204, 156], and such external services are not under the DBMS vendor's control.

4.5.3 Implementation Difficulty

The out-of-DBMS approach (Cerebro-Spark) is generally the easy one to implement. One naive implementation would be a `SELECT * FROM ...` query followed by some pipelines that feed the exported data to DL tools. The UDAF approach requires more effort to implement the MOP scheduler, wrappers for invoking DL tools, and pipelines that feed data to DL tools and return results to the DBMS. CTQ requires similar efforts as UDAF does, except its scheduler is asynchronous and slightly harder to implement due to concurrency in queries. DA requires the most effort because it needs to implement/port the whole DBMS table scan method, including locating, unpacking, and reading the pagefiles. If the table is compressed and TOAST-ed [20], then one must also implement/port the decompression and de-TOAST methods. Such work is ad-hoc, DBMS-specific, and may not even be viable for proprietary DB and pagefile formats. Simultaneously, because its execution is outside the DBMS, unified memory management is difficult, and it could interfere with other queries. As a result, more careful tuning and setting of configurations are required to implement DA.

4.5.4 Portability

Portability indicates how much code can be reused if one wants to change the underlying DBMS. The out-of-DBMS approach again excels in this area because it is almost agnostic to the DBMS and can usually be ported easily. UDAF approach is also portable as it requires only UDFs and UDAFs, which are supported in most DBMSs. Medium efforts are needed to export these functions to other DBMSs. CTQ is largely similar to UDAF, except it, in addition, requires the DBMS to support concurrent targeted queries. DA is the less portable option, as it is deeply coupled with the DBMS. Unless the target DBMS employs a similar physical storage layer, to port one existing DA implementation would be difficult.

4.6 Empirical Comparisons and Analyses

We will first thoroughly compare the end-to-end performance of all the described approaches and study the tradeoff space. Then we will study the effects of factors such as heterogeneous and AutoML (Hyperopt) workloads and model sizes. We will also evaluate the scalability of each approach. All of our source code, data, and other artifacts are available at [7]. We will test on both GPU-enabled and CPU-only environments. One might wonder how GPUs will be available in practice for users that operate traditional DBMS clusters. As per Greenplum team estimates [2], at least 80% of its customers continue using on-premise clusters, largely due to privacy and security concerns, especially in the government, financial and health care sectors. Such users are increasingly purchasing GPUs and connecting them to their Greenplum clusters for in-house deployment of DL workloads. In cloud-native DBMSs such as AWS Redshift, one can easily spin up GPU instances and connect them with the DBMS instances. Use of hybrid cloud and public cloud is also increasing. It is not uncommon to run POCs and tests in public cloud with rented GPUs, before purchasing GPUs for in-house production deployment.

Compared approaches. We compare Cerebro-Spark, UDAF, CTQ, DA (renamed to DA-Cerebro), and MADlib MA, which is included as a key baseline. Only for the end-to-end test, we also include PytochDDP, a fine-grained out-of-DBMS data parallel DL training framework; it relies on NCCL and MPI for communications. We have also combined DA with PytorchDDP (named as DA-PytorchDDP) so that it can work with DB data directly. For the Hyperopt tests in Section 4.6.2, we also include a system called Hyperopt-Spark, which is an out-of-DBMS task parallel model selection system.

Datasets. We use two large benchmark datasets: *ImageNet* [78] and *Criteo* [73]. We use the processing scripts and versions released as part of Cerebro [214, 1]. ImageNet contains 1.2M images with 1000 classes; it has an on-disk size of 250GB. Criteo has 100M data points, binary classes, and an on-disk size of 400GB.

Table 4.6. Runtimes and resource utilizations of end-to-end tests. Execution time and all utilizations are measured excluding ETL. Per-epoch time equals Execution time divided by number of epochs. Total network means the total amount of data transmitted during execution. We report disk read/write as per worker average. *These methods showed little to no disk reads because data has been cached in memory during the ETL process.

	Approach	ETL time	Exec. time	Epoch time	GPU util.	GPU RAM util.	CPU util.	DRAM util.	Tol. network	Per w. disk R/W
ImageNet	MA	2.8 h	42.6 h	4.3 h	56.8%	32.5%	2.3 %	3.1%	0.9 TB	12 GB/ 2 GB
	UDAF	2.8 h	48.5 h	4.9 h	49.9%	28.6%	2.2%	5.6%	0.8 TB	12 GB / 279 GB
	CTQ	2.8 h	45.1 h	4.5 h	56.2%	32.2%	2.5%	1.9%	0.6 TB	12 GB / 152 GB
	DA-Cerebro	5.4 h	23.0 h	2.3 h	70.5%	42.5%	2.8%	20.2%	0.6 TB	0.6 GB* / 0.3 GB
	Cerebro-Spark	4.4 h	23.9 h	2.4 h	65.1%	36.5%	11.2%	17.4%	1.1 TB	0.2 GB* / 2 GB
	PyTDDP	4.4 h	77.3 h	7.7 h	97.1%	13.1%	8.1%	14.7%	1900 TB	None* / 11 GB
	DA-PyTDDP	5.4 h	77.5 h	7.8 h	96.8%	13.2%	8.2 %	21.1%	1900 TB	None* / 1 GB
Criteo	MA	8.6 h	38.5 h	7.7 h	N/A	N/A	44.1%	2.3%	0.1 TB	1 GB / 2 GB
	UDAF	8.6 h	62.0 h	12.4 h	N/A	N/A	27.1%	2.3%	0.1 TB	1 GB / 38 GB
	CTQ	8.6 h	40.0 h	8.0 h	N/A	N/A	41.0%	1.9%	0.08 TB	1 GB / 22 GB
	DA-Cerebro	10.5 h	21.5 h	4.3 h	N/A	N/A	37.4%	28.5%	0.07 TB	0.2 GB* / 0.3 GB
	Cerebro-Spark	8.3 h	22.5 h	4.5 h	N/A	N/A	35.2%	28.5%	0.2 TB	0.2 GB* / 1 GB

Workloads. We use various DL model selection workloads with different degrees of heterogeneity for different tests. Please refer to each corresponding section for details. We use Adam [149] as the mini-batch SGD method for all tests.

Experimental Setup. We use one cluster on CloudLab [246] with 8 worker nodes and 1 master node. Each node has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, 1TB HDD, and 10 Gbps network. Each worker node also has an Nvidia P100 GPU. For tests with MLP on the Criteo dataset, we disable the GPUs to demonstrate the system's performance under CPU-only setting. All nodes run Ubuntu 16.04. We use GPDB 5.27, Spark 2.4.5, Cerebro figures/1.0.0, TensorFlow 1.14.0, Pytorch 1.4.0, CUDA 10.0, and cuDNN 7.4. Both datasets are randomly shuffled and split into 8 equal-sized partitions.

4.6.1 End-to-end Performance Study

We first present the end-to-end results for both ImageNet and Criteo. For ImageNet, we use two different neural architectures and a hyperparameters grid, yielding 16 training configs. For Criteo, we conduct a hyperparameter-tuning-only workload with also 16 training configs.

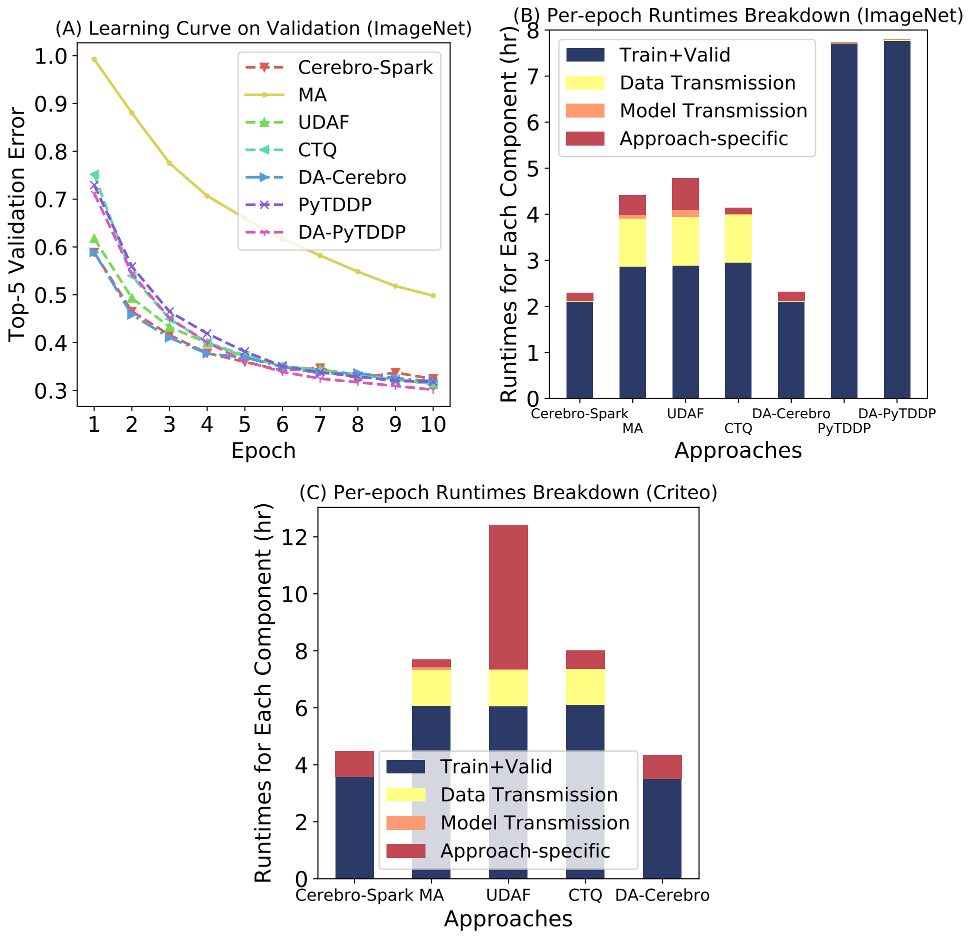


Figure 4.9. End-to-end tests results. (A): Convergence behavior on ImageNet. (B): Per-epoch breakdown of runtimes for each approach on ImageNet. (C): Per-epoch breakdown of runtimes for each approach on Criteo.

Table 4.5 offers the details. Such grid search-based model selection is standard in DL practice and still widely used by practitioners [50]. We compare our various architectural approaches with each other. MA is the baseline for this comparison.

For each different approach, separate ETL processes must be done beforehand. For UDAF, CTQ, and MA, ETL is in-DBMS preprocessing that packs the original data into byte arrays and buffers for the UDAFs to consume. For DA, ETL includes the above processing, plus accessing tables and TOAST pagefiles, de-TOAST, and loading into the main memory. For Cerebro-Spark, ETL consists of data exporting from DBMS and preprocessing to cast the data formats; we use a distributed Greenplum ETL tool `gpfdist` [279] for exporting and a customized program for preprocessing.

We examine the performance on multiple fronts: convergence, runtime, and resource utilization/cost including GPU/CPU, DRAM, network, and disk. Figure 4.9(A) demonstrates the convergence behaviors for ImageNet. All but MA converge to the same optima, as they are equivalent to sequential SGD. MA, on the other hand, has a convergence problem and learns much slower than the rest. We skip the convergence curves on Criteo for brevity's sake because all methods, including MA, have almost indistinguishable convergence behavior (reaching 99% accuracy quickly).

Table 4.6 summarizes the runtime performance and resource utilizations/costs.[3] MA, UDAF, and CTQ show close speed. They have identical reads, equal to the local on-disk pagefile size (12 GB for ImageNet and 1 GB for Criteo). After the first table scan the pagefile remains in the OS cache. MA is marginally the fastest among them, but note that it has poor convergence, as Figure 4.9(A) shows. CTQ is slightly faster than UDAF due to the removal of sub-epoch level synchronization barriers. The benefit is not obvious here, but we will drill deeper in Section 4.6.2. DA-Cerebro and Cerebro-Spark show the best performance and are close in runtime. This shows that one can achieve the same high performance as a SOTA out-of-DBMS DL approach while still operating on DB-resident data. The two data parallel methods PytorchDDP and DA-PytorchDDP are heavily bottlenecked by networking (over 1700x higher cost compared to other approaches). They both showed high GPU utilization only because they employ GPU for communication. They have less GPU memory consumption because how Pytorch differs from TensorFlow on memory management. They performed even slower on Criteo and were estimated to take over 16 days of runtime each, so we skipped these tests. DA-Cerebro, Cerebro-Spark, PytorchDDP, and DA-PytorchDDP showed higher DRAM usage because of caching. They also showed few disk reads due to preloading and caching during ETL; the writes are due to metadata

[3]Runtimes may not be directly comparable to figures in [214], as in this paper, we adopted newer model implementations and different on-disk file formats.

management. Note disk R/W for all approaches are not significant, and none of them is bound by the disk IO speed.

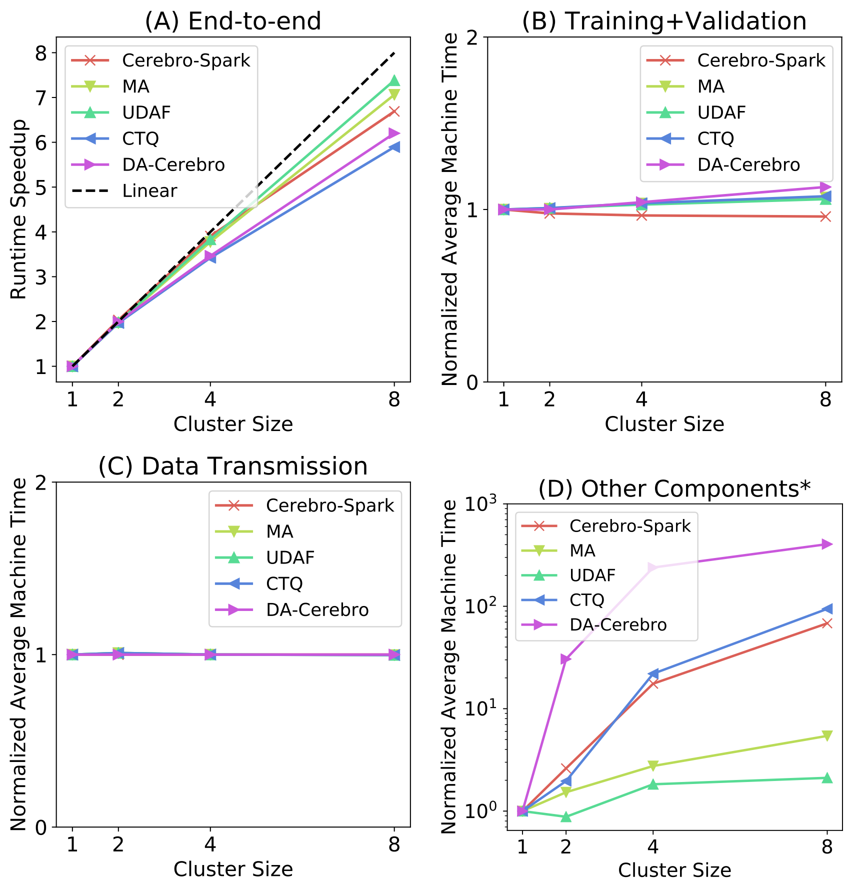


Figure 4.10. (A): End-to-end scalability plot, y-axis shows the speedups with respective to single-node runtime. (B - D): Per-epoch machine time for each component normalized against single-node. The machine time is averaged among all nodes. *Other Components: includes Model Transmission and Approach-specific as described in Section 4.6.1.

Profiling and breakdowns. To further investigate the root cause of performance differences, we take both datasets and profile every approach by running several more breakdown tests and calculating each execution component's runtime. Excluding the ETL time, Figure 4.9(B) presents results for ImageNet, and Figure 4.9(C) presents results from Criteo tests. We record per-epoch machine time compositions for each worker and take the average among them. Hence, the summations of the runtime numbers are close to but may not be identical to the end-to-end runtimes in Table 4.6, which are determined by the slowest worker runtime instead of the mean runtime. We break down per-epoch runtimes into four different components:

1. **Train+Valid**: time spent in the DL tools, including initialization and destruction of models, allocation and freeing of GPU memory, training and validation with GPU, etc. MA, UDAF, and CTQ are less efficient because these in-DBMS approaches invoke the DL tools through wrappers that cause extra overheads. For PytorchDDP and DA-PytorchDDP, since they overlap communication with computation, Train+Valid also includes time spent on model updating communications (we name these Model Transmission, introduced below). For this reason they showed very high Train+Valid time because they are bounded by networking.

2. **Data Transmission**: time spent on transmitting data to the DL tool from storage. For in-DBMS approaches, it also includes the data decompression time. This component is non-negligible for the 3 in-DBMS approaches due to data access overheads, while in the rest approaches, training data is cached in memory during ETL; this part costs little. Recall from Table 4.3 that Cerebro-Spark and PytorchDDP suffers a 2x storage blowup, while DA-based approaches do not.

3. **Model Transmission**: time spent on transmitting serialized models/gradients between workers. For MA, Model Transmission means model collecting and broadcasting; it means model hopping for the rest MOP-based approaches. UDAF and MA are not so efficient on this end because they require database joins for re-distributing models. We will further investigate this performance gap in Section 4.6.2. The two PytorchDDP approaches showed none on this front only because it is absorbed into Train+Valid.

4. **Approach-specific**: for MA, it is the time spent on averaging model weights. For the rest, it means sub-optimal scheduling and/or idling of some workers. Cerebro-Spark, CTQ, and DA-Cerebro use an asynchronous random scheduler and work better for heterogeneous

workloads. As for UDAF, the performance is affected by its synchronous round-robin scheduler. We further discuss on these two schedulers in Section 4.6.2.

Comparing the Criteo tests to ImageNet tests, we notice two significant differences: (1). Model Transmission time drops for MA and UDAF. The MLP model used in Criteo tests is smaller than the CNNs used for ImageNet. (2). The UDAF approach suffers more from idling in Criteo because the workload is more heterogeneous due to highly disparate batch sizes.

Overall, the MA approach shows unfavorable convergence behavior and the fine-grained data parallel approaches (DA-PytorchDDP and PytorchDDP) are heavily bottlenecked. MOP-based approaches largely dominate these two parallelization models. As for MOP, the in-DBMS approaches suffer from various overheads and are, in general, less efficient than the DA-Cerebro approach and the Cerebro-Spark approach. However, recall that runtime efficiency is not the only criterion for such a system, as we showed earlier in Table 4.3. There exists a tradeoff space and perhaps no universal optima to the question.

4.6.2 Drill-down Experiments

Scalability (strong scaling)

In this test, we evaluate the strong scalability. We used clusters with 1, 2, 4, 8 workers with ImageNet. We use a workload of 8 homogenous configs (4 learning rates, 2 regularization values, and ResNet50 architecture) trained for one epoch. All runtimes exclude ETL. Figure 4.10(A) presents the results.

All approaches show close-to-linear scaling. To better understand these behaviors, we further drill down each runtime component and evaluate their scalability separately. For this purpose, we collect the average machine time spent on each component from all workers varying cluster size; we then report them against the single node time. Figures 4.10(B-D) summarize the results. Flat lines indicate that the component's machine time is constant regardless of cluster size, therefore perfectly scalable. An increasing curve means sub-linearity, and vice versa.

Figure 4.10(B) and Figure 4.10(C) show that the Training+Validation and Data Transmission component scale almost linearly for all approaches. The Model Transmission part is minuscule in the end-to-end time, thus we report it collectively with the Approach-specific components in Figure 4.10(D). For DA-Cerebro, CTQ, and Cerebro-Spark, workers may idle relatively more when the number of models approaches the number of workers. The random scheduler they use can yield sub-optimal scheduling under such circumstances. [214] Hence they all show sub-linear scalability, especially when cluster size grows from 4 to 8. On the other hand, UDAF adopts a round-robin scheduler to emulate MOP, which happens to be optimal for this specific homogenous workload; thus, it shows better scalability. MA utilizes all workers and shows no idle time, but the model averaging cost still rises a little when the cluster size grows.

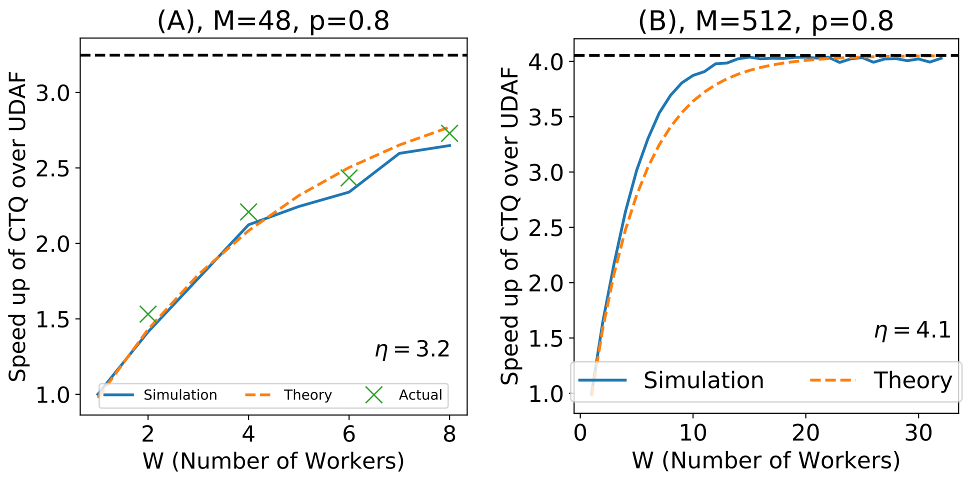


Figure 4.11. Heterogenous experiment. (A) Real experiments supplemented with simulation and theoretical results. $l_m/l_s = 8$ (B) Extreme scenario simulated. $l_m/l_s = 20$.

Async. MOP vs sync. MOP on Heterogeneous Workloads

To verify Equation 4.1 and Equation 4.2 proposed in Section 4.5.1 and prove the benefit of async. MOP over sync. MOP for heterogeneous workloads, we conduct the following experiments. We have one sync. MOP approach: UDAF; and among the 3 async. MOP approaches we pick CTQ, as the main difference between UDAF and CTQ is only the synchronization model of scheduler. Following the analysis in Section 4.5.1, let $\mathbb{M}$ be drawn from a Bernoulli distribution: $Pr(l_x = l_s) = p, Pr(l_x = l_m) = 1 - p$.

We then test with real experiments. The fast model is MobileNetV2 with batch size 128, while the slow model is NASNetMobile with batch size 4. We down-sampled 6% of ImageNet, so that the experiments can finish in reasonable amount of time (same experiments on the full datasets are estimated to cost over two months). Sampling might alter the ratios between constant overheads and components that scale with dataset size. However, since the constant overheads are minuscule, the sampling proves not to affect our conclusion. In Figure 4.11(A) we see the actual runs fit nicely with our simulation and theory. Furthermore, Figure 4.11(B) shows one simulated extreme scenario with a large workload and 32 workers to demonstrate the theoretical upper bound of the speedup. We refer interested readers to B for more simulations. Overall, these experiments verify that our theoretical bounds match with the actual runtime gaps. Meanwhile, we also show that the upper bound of speedup is determined by η. This indicates that CTQ can be a more efficient choice than UDAF when working with highly heterogeneous workloads and/or hardware.

Effect of model size on UDAF and CTQ

The size of models is typically orders of magnitude smaller than the size of training dataset. Thus, although model hopping time is proportional to model size, it is usually negligible in large-scale DL. However, this assumption may not hold for the UDAF approach because of the `JOIN` as explained in Section 4.5.1 Model hopping. We run a test to investigate model transmission cost with varying model sizes empirically. Our test shows that the CTQ approach imposes little to no bottleneck and is far less sensitive to the model size. However, the UDAF approach suffers more overheads on larger models. This confirms that the JOIN and storing models inside the DB can indeed cause some overheads, although this overhead is not too major (less than 10% in this case). The details of this test can be found in B.

Experiments with Hyperopt Workloads

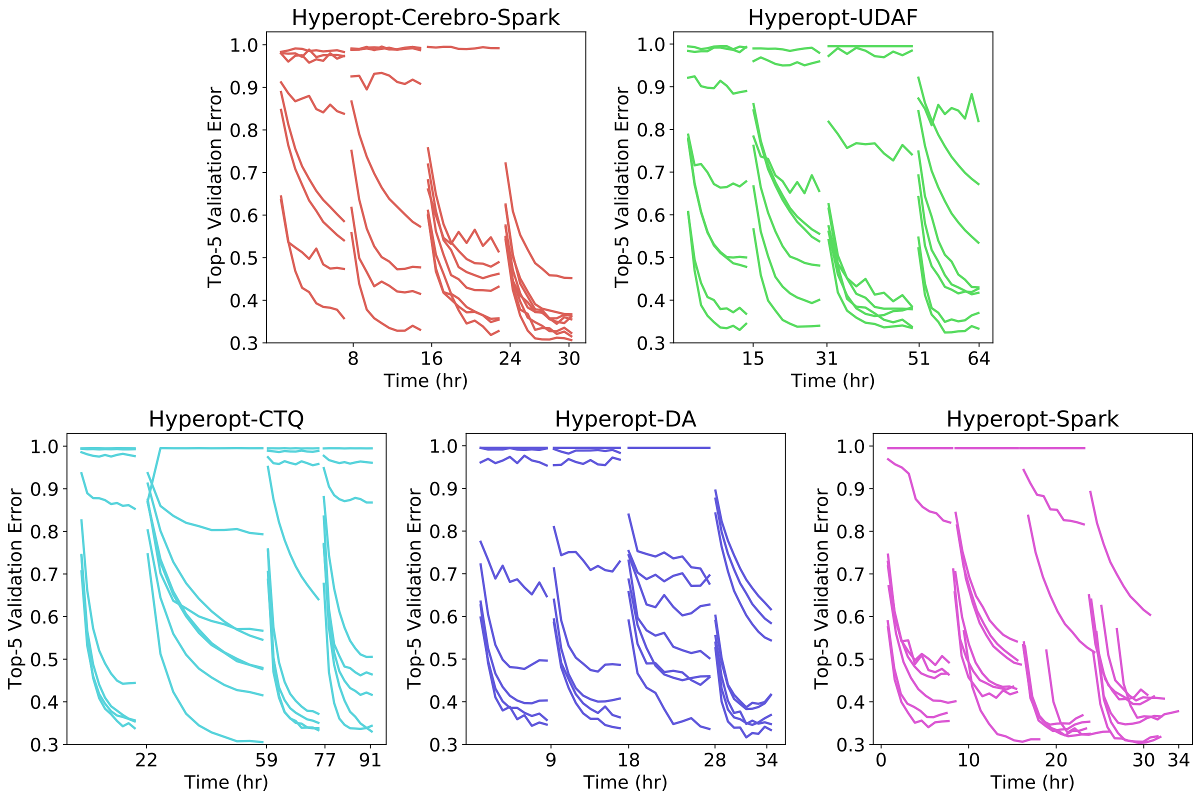


Figure 4.12. Hyperopt learning curves. Each diagram contains learning curves of all 32 model configs. Best val. errors achieved by each approach are within the margin: 0.31 (Cerebro-Spark), 0.33 (UDAF), 0.31 (CTQ), 0.33 (DA), 0.31 (Hyperopt-Spark).

In the end-to-end experiments we used a simple grid search workload. Now we evaluate the generality of workloads for the approaches. We use a model selection workload guided by Hyperopt (TPE algorithm) [41]. The parameter grid we use to sample model configs is as follows. Model: [ResNet18, ResNet34]; Learning rate: [10^{-5}, ..., 10^{-1}]; Weight decay: [10^{-4}, 10^{-6}]; Batch size: [16, ..., 256]. We also include a comparison to Hyperopt-Spark [129], a standalone task parallel model selection system. We set the number of model configs to 32 and degree of parallelism to 8. Figure 4.12 plots the learning curves. CTQ has ~50% higher runtime than UDAF; This is because MOP's random scheduler has a decreased runtime performance when the degree of parallelism is close to the number of workers [214] and showed in B. This issue can be largely mitigated by increasing the degree of parallelism. DA/Cerebro-Spark run similarly to Hyperopt-Spark, but the latter requires both data export and full data replication to each worker. Therefore it has a storage blowup of 9x, while Cerebro-Spark has 2x and DA has none in this case.

On GPU utilizations, the conclusion is consistent with those showed in Section 4.6.1. We have 32% (UDAF), 33% (CTQ), 44% (Cerebro-Spark), 44% (DA), and 45% (Hyperopt-Spark). The rest of the measurements are available in B.

Implementation Difficulty

Implementation difficulty is harder to measure quantitively. Following the discussion in Section 4.5.3, we now try to provide a more quantitive measurement in the form of lines of source code (LOC). The APPROACH (LOC) is as follows: UDAF (5866), CTQ (5939), DA-Cerebro (4230: 2764 for standalone Cerebro and 1466 for DA), and Cerebro-Spark (4338). Note these are counted for end-to-end implementation. UDAF and CTQ can largely share codebase: given UDAF, it requires only a few hundred lines to implement CTQ as well. Overall, DA and Cerebro-Spark take less code to implement than their counterparts UDAF and CTQ. However, this does not necessarily mean they are subjectively easier; as discussed in Section 4.5.3, the DA approach was much more time-consuming than the LOC number would otherwise suggest.

4.7 Conclusions

Through the paper, we comparatively and quantitively evaluated the fitness of various distributed deep net training schemes in existing data systems. We characterize the particular suitability of MOP for DL on data systems, but to bring MOP-based DL to DB-resident data, we show that there is no single “best” approach, and an interesting tradeoff space of approaches exists. We explain four canonical approaches and build prototypes upon Greenplum Database, compare them analytically on multiple criteria (e.g., runtime efficiency and ease of governance) and compare them empirically with large-scale DL workloads. Our experiments and analyses show that it is non-trivial to meet all practical desiderata well and there is a Pareto frontierOur results and insights can help DBMS and cloud vendors design better DL support for DB users.

Chapter 4 contains material from “Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches” by Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 14, Issue 10, July 2021. The dissertation author was the primary investigator and author of this paper. All of our source code, data, and other artifacts are available at https://github.com/makemebitter/cerebro-ds.

Chapter 5

LOTAN: Bridging the Gap Between Graph Data Systems and Graph Neural Network Workloads.

5.1 Introduction

In this chapter, we turn to the DL model selection and training problems on graph data. Graph data is non-Euclidean and has vastly different representations from the regular tabular or image data. To tackle the many graph data-driven applications and the irregular nature of graphs, DL methods, called Graph Neural Networks (GNNs), have been proposed. GNNs have drastically shifted the landscape of advanced graph analytics. They can provide powerful learned representations for graphs. In about a decade, GNNs have dominated many graph analytics leaderboards [124] for tasks ranging from lower-level ones, such as node classification and edge prediction, to graph-level tasks like graph classification or even graph generation. Applications span from video analytics [131], recommender systems [295, 306], drug discovery [179] and pandemic data analysis [299], to even crime prediction [264] with spatial-temporal graphs. Interest in GNNs is rising rapidly in many domains where data are naturally represented as graphs, such as social networks and molecular structures.

However, GNN models are tricky to scale [297, 113, 294], because of the sheer amount of computation and the immense memory pressure they exert on GPUs. A plethora of GNN systems

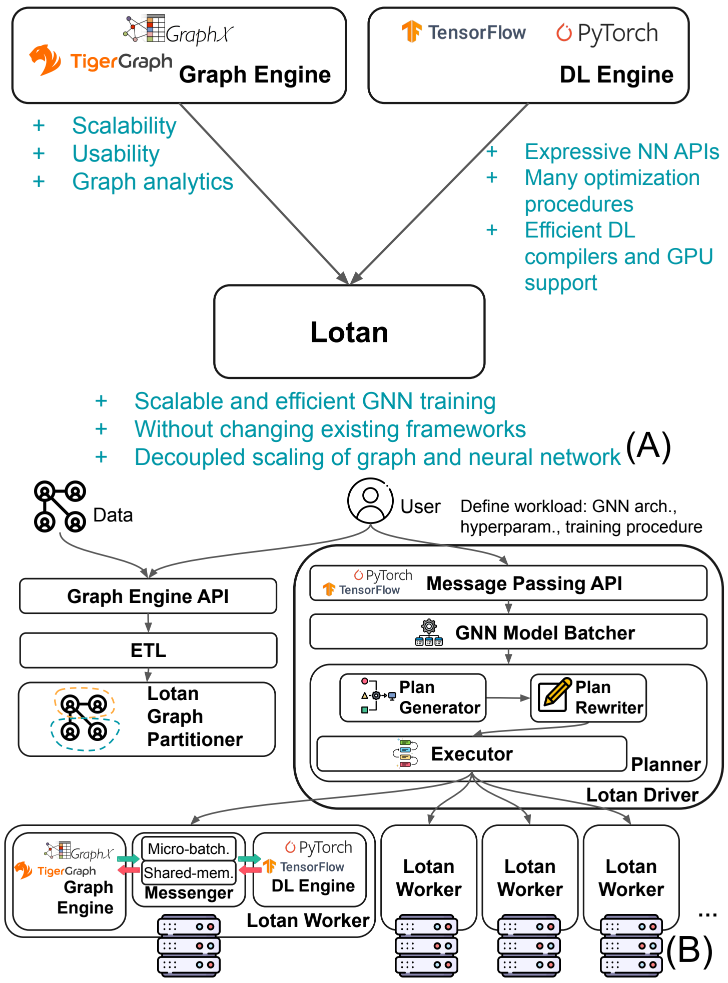


Figure 5.1. (A) Lotan bridges the gap between graph systems and DL systems. (B) The architecture of Lotan.

was proposed to tackle these challenges [285, 323, 93, 195, 186, 280, 135, 230]. They express GNN workloads primarily as advanced matrix multiplications and rely on GPUs for execution. When GPU memory is insufficient to host the entire matrices and the intermediate results, one either resorts to distributed processing [323, 135] and/or spilling techniques [135, 280] that load/offload data from GPU accordingly.

What makes GNN training so hard to scale, and why do we need these dedicated systems for GNNs? First, graph data are irregularly shaped and non-IID, differentiating them from regular IID data modalities such as text and images, for which the state-of-art DL frameworks were designed. To tackle the data scalability issues, most DL frameworks employ distributed data-parallel schemes [21, 257]. However, data parallelism does not directly apply to graph data: graph partitions are not independent, and the training process involves cross-partition communications, depending on the input graph structure. Second, neural network backpropagation requires caching intermediate data during forward propagation. Depending on the graph data, these intermediates could be huge in size. Unlike models such as CNNs or Transformers designed for IID data, where input shape is often normalized and uniform, GNNs are highly input-dependent. They are tough to accommodate, as workloads are highly versatile and vary significantly in scale. Third, the "neighborhood explosion" and the over-smoothing problems [66, 201, 68] are also tricky to bypass; data dependency grows exponentially as the number of GNN layers grows, posing challenges to both scalability and efficiency.

In this chapter, we make a critical observation that **many of the GNN's challenges are, in fact, challenges of managing, moving, and handling the underlying graph data.** Nonetheless, existing custom GNN systems mix and couple the graph data and DL challenges. We observe several shortcomings of this worldview: first, many of these systems "reinvent the wheel" of much work done in the database world on scalable graph analytics engines. Second, they often tightly couple the scalability treatments of graph data processing with that of GNN training, resulting in entangled, complex problems and systems that often do not scale well on one of those axes. GNN workloads, though drastically different from regular DNN workloads in data access patterns, are not too far away from non-NN graph analytics such as PageRank. As pointed out by prior work [195], most of the popular GNNs can be expressed under extended versions of graph programming models such as Gather-Apply-Scatter (GAS). Scaling "shallow" graph analytics is not a new topic: many graph data systems were designed for that purpose [198, 106, 103, 105]. However, to our knowledge, none of these systems provide general GNN support, nor do they handle DL operations, which, nowadays, are better reserved for frameworks such as TensorFlow and PyTorch. It would be prohibitively labor-intensive to build systems of their generality and performance from scratch. Furthermore, implementing native NN support within graph systems would lead to a similar problem of reinventing the wheel of DL systems research. Therefore, both stacks of software are needed: graph systems for graph challenges and DL systems for DL challenges. As Figure 5.1(A) explains, our work aims to bridge this gap.

System Desiderata. We envision a scalable GNN system with the following desiderata: (1) *Decoupled Scaling*: Scale graph and neural network parts by reusing existing industrial-strength tools. (2) *Usability*: Retain the ease of specification APIs of both graph and DL tools. (3) *Non-disruptive Integration*: No changes to the internal code of those tools. (4) *Speed and Accuracy*: Fast runtimes without sacrificing DL accuracy.

In this chapter, we seek to answer a fundamental systems question: *How far can we go by pushing existing systems' limits without modifications?* We propose a novel information system architecture for scalable GNN training with the **decoupling of graph and neural network**. Much like the famous **decoupling of compute and storage** in cloud computing, this decoupling enables us to tackle each side individually and allows them to scale independently. We carefully pick apart the graph and neural network dataflows in GNN training and re-imagine them as a "query plan" in our new intermediate-level *global operator graph.* We dispatch the execution plan to an existing graph analytics engine and a DL framework without modifying their internal code. We built a distributed prototype system we call Lotan.

Overview of Lotan. Figure 5.1(B) illustrates our system architecture. The user interacts with Lotan through the APIs to specify their GNN workload. Our Planner then compiles it into graph and neural network operations and dispatches them to their separate execution Engines, which are existing graph and DL systems. Our Messenger handles the coordination and communications between the Engines. This way, Lotan can preserve all functionalities of the execution Engines, especially the graph data management functionalities such as graph manipulation, partitioning, and other non-NN graph analysis methods.

Additionally, Lotan provides a series of system optimizations to increase the runtime performance. The most important two are GNN-centric graph partitioning and GNN model batching.

GNN-centric graph partitioning. Distributed graph processing naturally comes with the problem of graph partitioning, which can dramatically affect the efficiency as sub-optimal partitioning leads to a huge amount of network communications. To this end, we propose a GNN-centric graph partitioning scheme and the corresponding Reverse Graph Propagation execution scheme for GNN training. Our method works by taking account of the asymmetry in data size between forward- and backward propagation. We will describe it in detail in Section 5.5.1.

GNN model batching. GNNs, like other DL methods, require extensive hyperparameter tuning, which involves training multiple models on the same dataset. These models overlap extensively in their data access patterns, and opportunities exist because data access is quite costly for GNNs. We propose the first GNN Model Batching technique to improve GPU utilization and reduce runtime for GNN model selection workloads. As far as we know, Lotan is the first system to optimize for the GNN model selection/hyperparameter tuning workloads and the first to explore model batching for GNNs. We will introduce it in Section 5.5.2.

Overall, we make the following technical contributions:

- To the best of our knowledge, this is the first work to bridge the gap between existing graph data systems and DL systems and the first to formally decouple the scaling of graph and neural networks in GNN training. Lotan expands design freedom for GNN researchers and practitioners.

- We re-imagine large-scale GNN training from a data management standpoint and unpack the dataflows into a "query plan" representation. We then devise novel query rewriting and optimization techniques to improve scalability and efficiency.

- We propose one of the first GNN-centric graph partitioning schemes to reduce graph node replication and communications during GNN training.

- Furthermore, Lotan is the first GNN system to treat model selection workloads holistically and explore model batching techniques to improve training throughput.

- We perform an extensive evaluation to compare Lotan with prior industrial-strength systems and study the impact of our optimizations. The empirical results validate our system's higher scalability and competitive time-to-accuracy performance on multiple workloads.

5.2 Background

5.2.1 Graph Neural Networks

Graph Neural Networks (GNNs) are neural networks on graph data. In a nutshell, a GNN always tries to summarize the graph structure and/or the graph properties into a compact numerical representation called embeddings. GNNs can be categorized into spectral-based and spatial-based methods [297]. Spectral-based methods have roots in graph signal processing and rely on the graph Laplacian and Fourier transform for generating embeddings. Spatial-based methods are typically the applications of neural networks such as RNN, CNN, and GAN on graph data, with modifications to account for graph structure. The spatial-based methods are the more popular of the two categories [297] and will be the main focus of our system.

It is important to note that a GNN model can be ultimately expressed as a combination of graph processing (in the form of a modified Gather-Apply-Scatter programming model [195]) and DL operations. This is the basis of how our system attacks the problem; we compile a GNN training task into a global operator graph composed of graph operators and neural network operators and use existing systems for execution. More details on these concepts are in Section 5.3.2 and Section 5.4.

5.2.2 Distributed Graph Processing

GNN workloads are still a form of graph processing/analytics because they resemble many classical problems and share very similar data access patterns. To tackle the many similar challenges, non-GNN graph data systems [272, 249, 44, 191, 106, 103, 80] rely on distributed processing, and a critical problem is graph data partitioning.

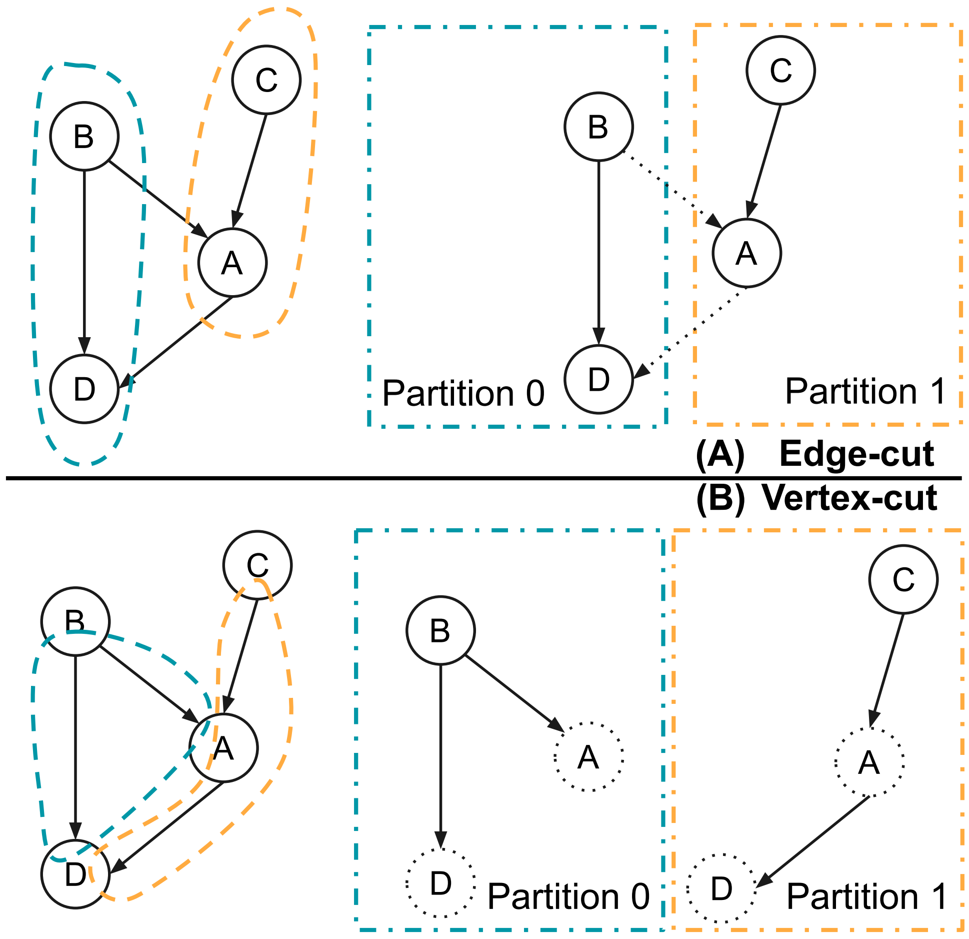


Figure 5.2. Two graph partitioning schemes.

There are two major graph partitioning schemes: edge-cut and vertex-cut. It is beyond the scope of this paper to fully cover the entire landscape of graph partitioning, so we only introduce the bare minimum background before we propose our own GNN-centric graph partitioning scheme in Section 5.5.1. Interested readers are directed to other literature on graph partitioning [56].

Edge-cut. Edge-cut partitioning affixes the location of vertices, and the edges at partitioning boundaries are replicated (or need to be remotely fetched when needed). Figure 5.2(A) illustrates this scheme. In Gather-Apply-Scatter workloads, the messages generated at vertices are sent across the partitioning boundary, resulting in cross-partition communications.

Table 5.1. Comparison with prior art on key capabilities.

	License	GPU	Distributed	Sampling	Memory Hierarchy
Lotan	Open	✓	✓	Full	Disk-aware
DGL/DistDGL [285]	Open	✓	✓	Both	GPU-only
AliGraph/graph-learn [326]	Open	✓	✓	Mini-batch	GPU-only
PSGraph [138]	N/A	✓	✓	Mini-batch	GPU-only
GraphScope [301]	Open	✓	✓	Mini-batch	GPU-only
Sancus [230]	Open	✓	✓	Full	GPU-only
PipeGCN [281]	Open	✓	✓	Full	GPU-only
Dorylus [270]	Open	✗	✓(Serverless)	Full	N/A
ROC [135]	Open	✓	✓	Full	DRAM-aware
P^3 [98]	N/A	✓	✓	Mini-batch	GPU-only
DeepGalois [122]	N/A	✗	✓	Full	DRAM-only
Pytorch Geometric [93]	Open	✓	✗	Both	GPU-only
NeuGraph [195]	N/A	✓	✗	Full	DRAM-aware
PaGraph [186, 34]	Open	✓	✗	Mini-batch	DRAM-aware
MariusGNN [280]	Open	✓	✗	Mini-batch	Disk-aware

Vertex-cut. Vertex-cut partitioning is the alternative to edge-cut; it focuses on the edges and fixes their locations. As a trade-off, the vertices at the boundaries need to be replicated or at least remotely fetched when needed.

Note that a certain amount of cross-partition communication or data replication is inevitable, depending on the quality of the graph partitioning algorithm and the characteristics of the underlying graph. Graphs typically have much more edges than vertices in the real world. Therefore vertex-cut partitioning, which avoids edge replications, sometimes are more favorable in practice [106, 191, 247] for non-GNN workloads. Although a plethora of graph partitioning algorithms exists [56], they are seldom designed for GNN workloads. As a result, there is room for improvement. We will dive deep into the characteristics of GNN workloads in Section 5.5.1 and subsequently propose our graph partitioning scheme on top of the vertex-cut scheme.

5.2.3 GNN Training Systems

Plenty of systems have been proposed to tackle the efficiency and scalability challenges brought by GNNs. Generally speaking, there are two main camps within GNN system research: first is the scalability camp, which aims to tackle the scalability issues of full-batch GNNs [135, 326, 270]; they are usually distributed systems and focus on providing the capability to run GNN workloads that fail on other systems. Second is the efficiency camp, which mainly focuses on runtime speed and usually does not address scalability issues; they often assume that the entire workload can comfortably fit in GPU memory/main memory [195, 186, 205, 230].

We summarize the comparisons between these systems in Table 5.1, and we will discuss them in more detail in Section 7.3. We evaluate these systems by several axes: (1) License, whether the system is open-source and usable for tests. (2) GPU, whether the system has GPU support. (3) Distributed, whether the system support distributed processing. (4) Sampling, whether the system targets full-batch or mini-batch GNN training. (5) Memory Hierarchy, whether the system is secondary storage aware. We leave the performance tests and numbers to Section 5.7. In this work, we argue that many of these systems are reinventing the wheel with custom-built graph data systems and are still facing scalability issues with larger datasets or models. Lotan is more closely related to the scalability camp, but it differs from the prior art in technical contributions and our architecture design, which can utilize existing, established systems.

5.3 GNN APIs and Programming Model

5.3.1 GNN Interface

The first system issue we need to address is *how do we express a GNN model in a standardized way?* One of the most common and general abstractions is known as the Message Passing interface [102]. It is also widely adopted in GNN system literature and the de-facto standard. The Message Passing interface defines a GNN using *update rule*, an equation that tells us how to update a graph node's embedding:

$$\mathbf{h}_v^k = \psi(\mathbf{x}_v^k, \bigsqcap_{u \in \mathcal{N}(v)} \phi(\mathbf{h}_v^{k-1}, \mathbf{h}_u^{k-1}, \mathbf{x}_{e_{vu}})), \tag{5.1}$$

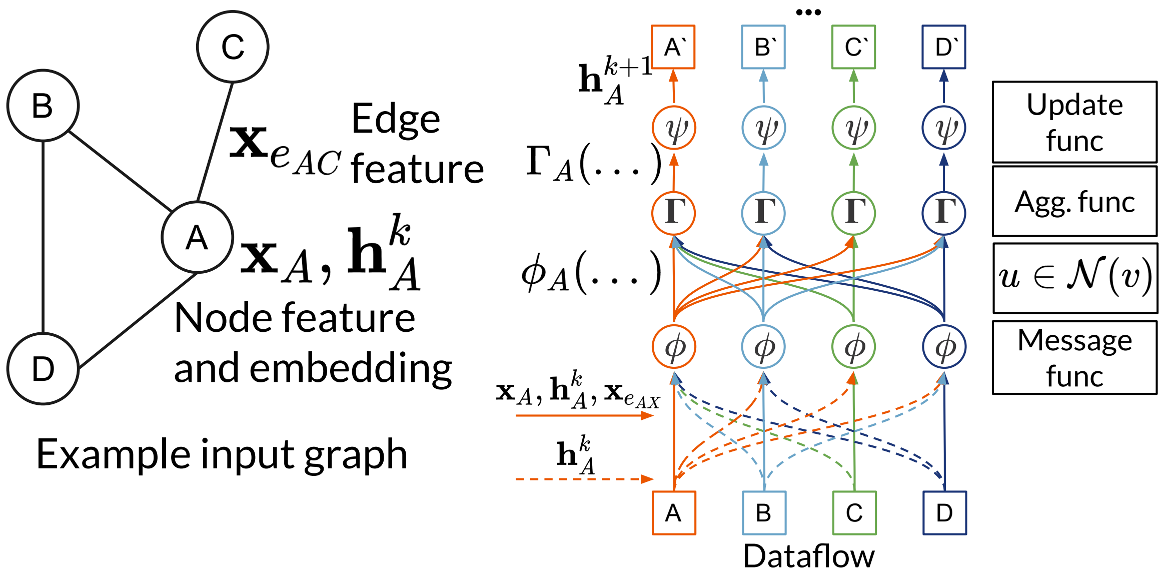


Figure 5.3. (A) An example input graph to a spatial-based GNN. (B) Dataflow diagram of a message passing GNN.

where ψ, ϕ, Γ are potentially learnable and differentiable functions, Γ is further required to be commutative and associative. ψ is called the update function, ϕ the message function and Γ the aggregate function. Note Equation 5.1 covers primarily spatial convolutional GNNs, on which we focus. Some other forms of GNNs, such as spectral-based methods [297], cannot be easily and efficiently expressed in the same framework. We leave the question of how to support those GNNs in future work.

Equation 5.1 is the interface we expose to the user. They will need to define the three functions ψ, ϕ, Γ using APIs and operators provided by the system to be introduced next. Depending on the nature of these functions, our system can do plan rewriting and optimizations to boost performance. More details are in Section 5.4.2. Figure 5.3(A) shows an example input graph and Figure 5.3(B) shows the conceptual dataflow of a GNN being learned on it.

Batched Message Passing. In practice, we find it much more beneficial to rewrite Equation 5.1 in a batched and vectorized format, especially for better utilization of GPU:

$$\mathbf{H}_v^k = \Psi(\mathbf{X}_v^k, \overset{*}{\boldsymbol{\Gamma}} \Phi(\mathbf{H}_v^{k-1}, \mathbf{H}_u^{k-1}, \mathbf{X}_{e_{vu}})), \tag{5.2}$$

where $\mathbf{H}_v^k, \mathbf{X}_v^k, \mathbf{H}_v^{k-1}, \mathbf{H}_u^{k-1}$, and $\mathbf{X}_{e_{vu}}$ are all matrices which are batched forms of their corresponding vectors, they have shape $B \times \_$, where B is the batch size and $\_$ is the dimension of the respective vectors. $\Psi, \Phi, \boldsymbol{\Gamma}^*$ are the batched (vectorized) form of functions ψ, ϕ, Γ.

5.3.2 Lotan's Internal Programming Model

It is not obvious how to parallelize Equation 5.1, mainly due to the neighborhood aggregation steps that are only native to graph processing systems. Some prior work re-cast the equation into bulk linear algebra [285, 323, 93, 326]. However, they often encounter huge scalability issues due to the potentially colossal graph size and the resulting sizes of matrix multiplications. On the other hand, it is not unfamiliar to see existing GNN systems using Gather-Apply-Scatter (GAS) abstraction or its extension [195] to translate Equation 5.1 into an executable plan. However, none of these abstractions capture the potentially costly data transfer operations between the graph and DL execution Engines that Lotan relies on. We need to account for these operations correctly so that we can better understand the problem. Toward this goal, we propose a new programming model that roots in the decoupling of compute and storage and the decoupling of graph and NN. Our abstraction involves three main stages: (1) Graph propagation with Scatter-Gather-Collect, (2) DL propagation with ApplyEdge-Aggregation-ApplyVertex, and (3) Pipe and Join. To implement these operators, we build them upon existing operators in the Graph and DL Engines.

Scatter-Gather-Collect. During this stage, the Graph Engine does a regular Scatter and Gather as in GAS for the graph propagation portion of a GNN. Instead of Apply in GAS, here it is followed by a Collect operation, where the Graph Engine, depending on the specifications of the GNN, collects and packs relevant data to hand over to the DL Engine. This arises because

GNN operations are placed on two different Engines in our system. The DL Engine operations (such as aggregation) have data dependencies up to the Graph Engine to resolve. The Graph Engine then needs to collect all the data from each graph node and their neighboring nodes and sends them to the DL Engine. Conceptually, this stage is primarily for implementing the neighborhood scope $u \in \mathcal{N}(v)$ in Equation 5.1.

ApplyEdge-Aggregation-ApplyVertex. In this stage, the DL Engine receives data from the Graph Engine and applies the GNN functions on the data. ApplyEdge implements the per-edge function ϕ; similarly, ApplyVertex implements the per-vertex function ψ. Aggregation implements the neighborhood aggregation function Γ.

Pipe and Join. We need operations for data transfer at the Graph and DL Engine's boundary. From the Graph Engine to DL Engine, we need a Pipe operation that, as the name suggests, pipes data to the DL Engine and the results back to the Graph Engine. Then within the Graph Engine, a Join operation is needed to incorporate the data, as the order of data may not be preserved during the Pipe. We will cover these in detail in Section 5.4.3.

One important note is that this separation of stages is not fixed; some operations can be eliminated, some can be re-ordered, and some can be pushed down. We will explore all these opportunities for optimizations in Section 5.4.2.

5.3.3 Global Operator Graph and Execution

With all the introduced abstractions, we can now compile an entire GNN training workload into a global operator graph with the operators mentioned above. A Planner, to be discussed in Section 5.4.2, will generate this graph from the user input expressed in the GNN message-passing interface.

Figure 5.4 shows the full operator graph for end-to-end GNN training, using the operators defined in Section 5.3.2. Data (embeddings during the forward-propagation, and gradients during the back-propagation) is sent back and forth between the Graph Engine and DL Engine. The Graph Engine is in charge of the graph aggregation by running Gather-Scatter-Apply under

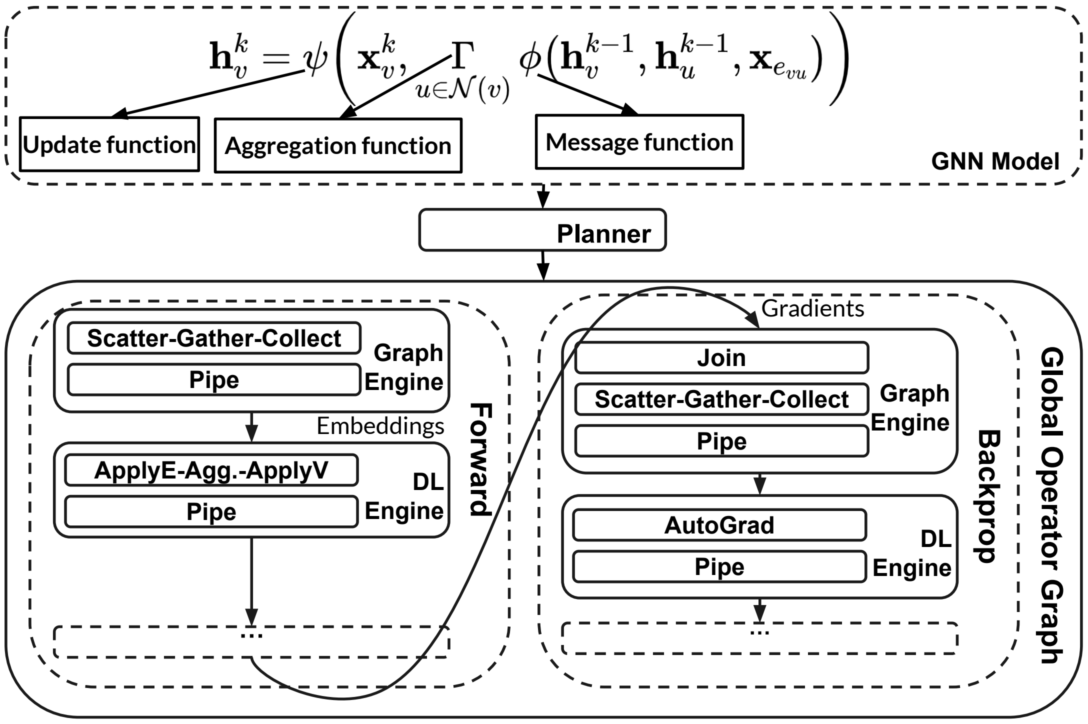


Figure 5.4. Global operator graph of end-to-end GNN training.

the hood for both forward- and back-propagation and collects all the necessary data for the DL Engine to consume, represented by the Collect operator. During the forward-propagation, the DL Engine handles the ApplyEdge, Aggregation, and ApplyVertex functions and subsequently does back-propogation with their AutoGrad capabilities. Both Engines run independently and are unaware of each other. They can run on the same set of machines, and the operators are parallelized independently. To coordinate the Engines and to provide a bridge for data transfer, we build a Messenger component for our system, to be introduced in Section 5.4.3.

5.4 System Architecture

Lotan has 3 main components: (1) External Engines, which are existing graph processing systems and DL frameworks **without modifications**. (2) Planner, where Lotan creates and optimizes the execution plan of a GNN training workload. (3) Messenger, where Lotan reconciles the Graph Engine and the DL Engine and facilitates efficient data transmission between them.

5.4.1 External Engines

These engines are what Lotan relies on and improves on for tackling many scalability challenges of GNN training. We only use these engines' public interfaces and treat them as black boxes. This way, we use them without modifications and drastically increase the portability and generality of Lotan while preserving all the features provided by both Engines.

Graph Engine. The Graph Engine is an external graph data system that Lotan relies on for graph-related operations and scalability challenges. It can be a graph processing system or a graph DBMS, as long as it provides public interfaces for (1) Gather-Apply-Scatter (GAS) operators. (2) Operations that export data to external systems. Additionally, it should provide scalable solutions for large-scale graph analytics. Often, such Engines conveniently provide various data system features such as data partitioning and distribution, fault tolerance, memory management, and disk spilling. Most of today's graph analytics systems/graph DBMS meet these criteria. Examples include Spark's GraphX [106], Giraph [103], TigerGraph [80], and Neo4j [219]. We choose GraphX for our prototype because, first, it is open-source software and has an active user community. Second, it is easy to use and piggybacks on the popular and familiar Spark ecosystem. Our approach is general and easily applicable to other graph analytics engines.

Deep Learning Engine. To handle the challenge from the neural network part of a GNN, we adopt an external DL system/framework. We use this system for forward propagation activation computing and back-propagation gradient computing with their autograd capabilities. By using an existing DL framework, we automatically make available the rich DL libraries and GPU support such a framework comes with. Furthermore, these systems can offer an out-of-box solution for distributed model training via their data-parallel capabilities [257, 177]. TensorFlow and PyTorch are both prominent examples of such systems and both are applicable. We pick PyTorch due to its dominant popularity in the GNN community.

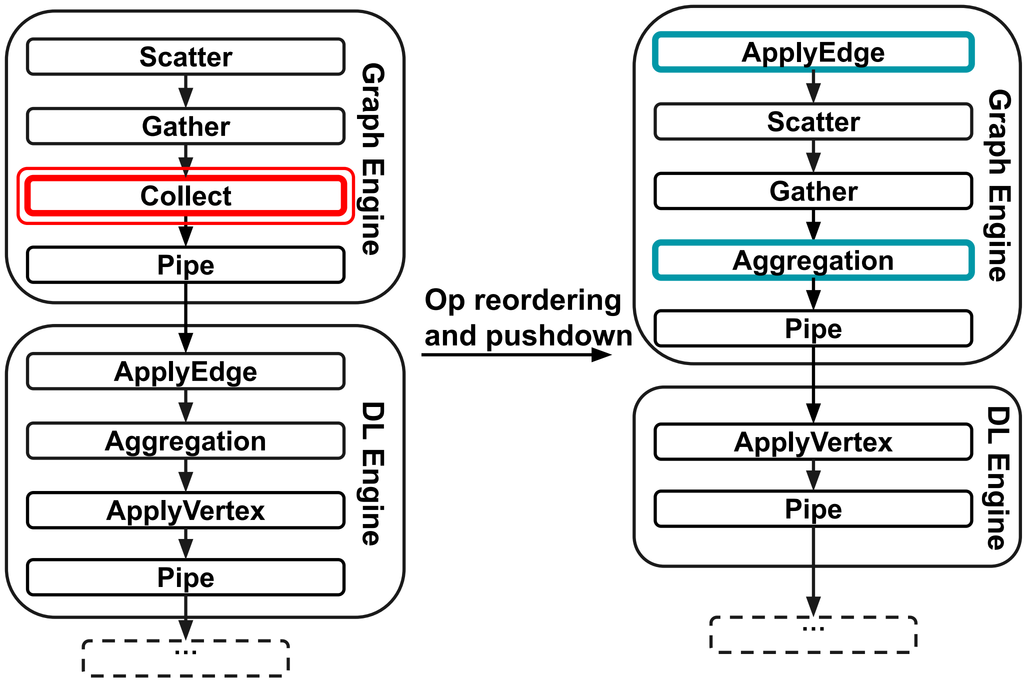


Figure 5.5. An example of plan rewrites. Note the Collect operator is rewritten with ApplyEdge and Aggregation.

5.4.2 Planner

At the heart of Lotan is the Planner, inspired by query planners in database research. Close to the concept of a DBMS query optimizer/planner, we need to weigh the potential query plans and choose the optimal one. The general idea is to assign relative costs for each stage of the execution and then determine the final cost estimate. However, in this case, the plan search space is much more limited, and it is favorable to do operator pushdowns whenever possible. Therefore, we find simple heuristics sufficient and no sophisticated cost estimation is needed. To complete the study, we still try to model the costs, but primarily for curiosity and a deeper understanding of the problem. We will also verify some of the observations from our cost models with experiments in C.

Plan Generation and Rewrites. Plan generation is usually trivial, as GNN training comprises mostly sequential stages, as Figure 5.4 shows. Opportunities for optimization exist;

depending on the nature of the GNN model, the execution plan can be rewritten. We only consider the two most obvious cases of plan rewriting: operator reordering and pushdown.

Equation 5.1 gives the most general definition of a GNN, and Figure 5.3(B) is the most stringent ordering of the three functions from the Message function to the Aggregation function to the Update function. However, because all these functions can be neural networks, they can only be handled by the DL Engine. Both the Message and Aggregation functions require neighbor information; we will also have to collect all the edges, features, and embeddings in the Graph Engine and ship them to the DL Engine. For this general case, our operator graph writes as Figure 5.4. This is an expensive plan due to the Collect operator and the size of data movement between the two Engines. However, if the Message and Aggregation functions are both unparameterized and therefore do not require training, we can push down these functions to the Graph Engine and drastically save costs. Figure 5.5 illustrates this scheme. We will test plan rewrites with experiments in Section 5.7.2 and see that it contributes to substantial performance gains.

Cost Estimation. To calculate the costs of a plan, we first evaluate the costs of individual stages respectively, then aggregate them together. A stage is defined as the sub-operator graph between two boundaries of data movement. The costs are estimated using: (1) the data graph's information, such as the number of nodes and vertices and the average degree. (2) specifications of the GNN, such as the number of layers and the number of parameters involved in the neural network. (3) The DRAM and GPU RAM limit, network/disk bandwidth, and the number of concurrent CPU threads available (degree of parallelism). Due to space constraints, we will highlight the main observations in Section 5.6, but leave the details to C. Within the cost models, a few factors are situation-dependent and, therefore, cannot be very well estimated. One can resort to logs of past runs of the same model and graph for more accurate costs.

5.4.3 Micro-batch Processing and Messenger

One critical question of utilizing existing systems is how to reconcile them; each comes with its input/output interfaces, data formats, memory layouts, and other specifications. Further, the DL Engine heavily favors batched data input for higher utilization and throughput, while the data comes off the Graph Engine as streams to reduce memory footprint. This means we must convert the data stream to and from data batches. We also need to keep the order of data consistent during both the forward pass and backward pass stages.

To our best knowledge, this is the first time the data movement issues between graph data systems and DL systems are being studied. We adopt and synthesize existing techniques and optimizations to solve the novel problems mentioned above. We build a component called Messenger. We apply a series of system optimizations to the Messenger: It uses non-blocking, async sockets and shared memory to communicate with the DL Engine for overlapping computation with communication and to reduce throttling. The details of this component can be found in C.

5.5 System Optimizations

5.5.1 GNN-centric Graph Partitioning and Reverse Graph Backpropgation

Graph partitioning is a vital part of distributed graph processing, as it dramatically impacts the volume of data replication and communications. Most existing graph partitioning schemes are not designed with GNNs in mind, resulting in suboptimal performance. We propose a novel graph partitioning and training execution scheme for GNNs, named Reverse Graph Backpropagation (RGB). This technique applies to vertex-cut-based Graph Engines [106, 105] such as GraphX, which we use for prototying. Our method is based on two key observations: first, neural network training consists of a forward- and a back-propagation phase; the two phases have inverted dataflow. Second, during GNN training, graph node data are updated, and the data size changes between phases, which leads to an asymmetry in replication costs.

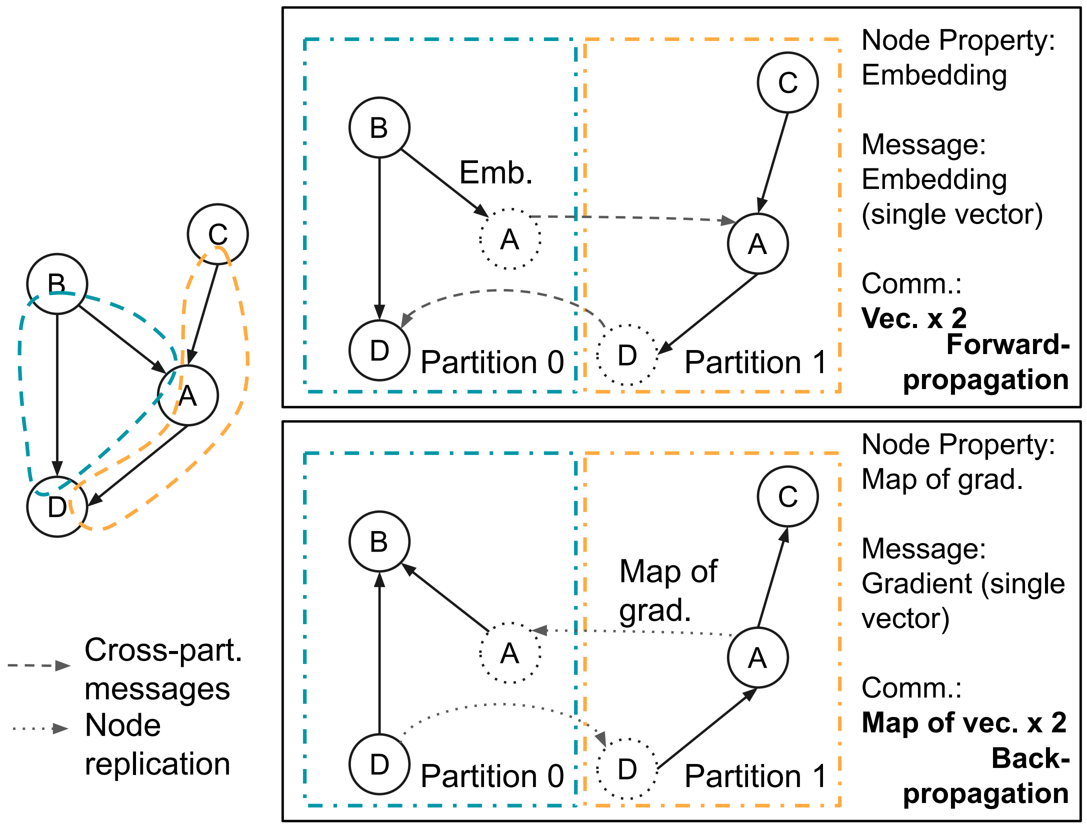


Figure 5.6. Regular 1D source hash partitioning and dataflow.

We start from the well-accepted hash-based 1D edge partitioning [247], where we hash partition all the nodes and then colocate all edges based on their sources. Figure 5.6 illustrates the strategy. During forward propagation, each node's property and messages it sends are its node embeddings, which are 1-dimensional vectors. No node replication happens, but two cross-partition messages take place. During the back-propagation, the data flow is inverted. However, each node updates its properties to gradients returned from the DL Engine (such updates happen in-partition and do not incur cross-partition communications). These gradients are hash maps of vectors and, compared to the embeddings, are d (node degree) times larger. For a realistic graph, the average degree can easily be around 100. Because dataflow is inverted and the partitioning is not, heavy cross-partition communications would occur.

To address this performance issue caused by the asymmetry, we propose our novel GNN-centric Graph Partitioning scheme and the way to backpropagate through it, described as follows:

1. Create a reverse graph (each edge reversed) from the original graph.
2. Do a regular hash partitioning on the reverse graph: first, hash partition all the nodes and place them; second, partition the edges based on their sources so that all edges originating from the same source colocate in the same partition.
3. Finally, we partition the original graph's edges in the same manner but keep the node partitions generated from the reverse graph.
4. We run the forward propagation as usual on the original graph. However, we run the back-propagation on the reverse graph.

This way, there is drastically reduced communication during back-propagation, where the most significant bottleneck could arise. Depending on the circumstances, communication costs might increase for forward propagation but are offset by back-propagation savings. We keep the node placements consistent between phases, otherwise extra cross-partition communication will occur. Figure 5.7 illustrates our approach. Regarding cross-partition communications, we only have single vectors instead of hashmaps of vectors. The example shows a directed graph, but the same logic still applies to undirected graphs.

5.5.2 GNN Model Batching

GNNs, like any other neural network, rely on careful and extensive hyperparameter tuning for the best accuracy performance. Consequently, the workloads are often multiple-model explorations. Each model has its different set of neural network hyperparameters. When running hyperparameter tuning workloads, existing systems take a sequential approach: training them one-by-one. Figure 5.8(A) shows it. There is wasted potential for improvements: First,

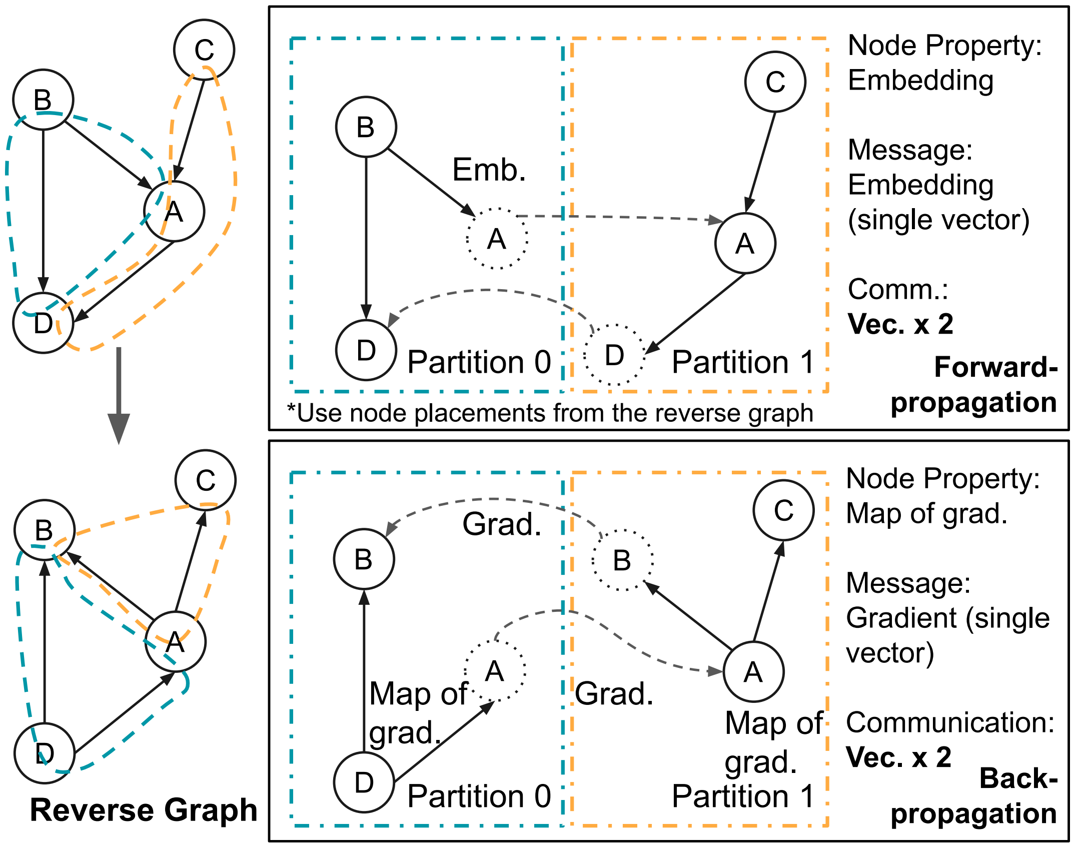


Figure 5.7. GNN-centric Graph Partitioning and dataflow.

models in a hyperparameter search workload share identical data access patterns, and re-using these routines can amortize the overheads. Second, many GNN workloads have relatively low neural network components, often leaving the GPU underutilized. For DL methods on IID and Euclidean data, many systems [214, 319] have been developed to optimize for model selection workloads. However, these techniques do not apply as they assume IID data.

To address these issues, we propose GNN Model Batching. Model batching [217] is a technique to increase GPU utilization for IID models. To our best knowledge, we are the first to explore the same possibility for GNNs. We devise a model batching scheme to combine the models within a hyperparameter search workload. Figure 5.8(B) depicts it. We run multiple models simultaneously on the model-batched version of the regular graph and NN operators.

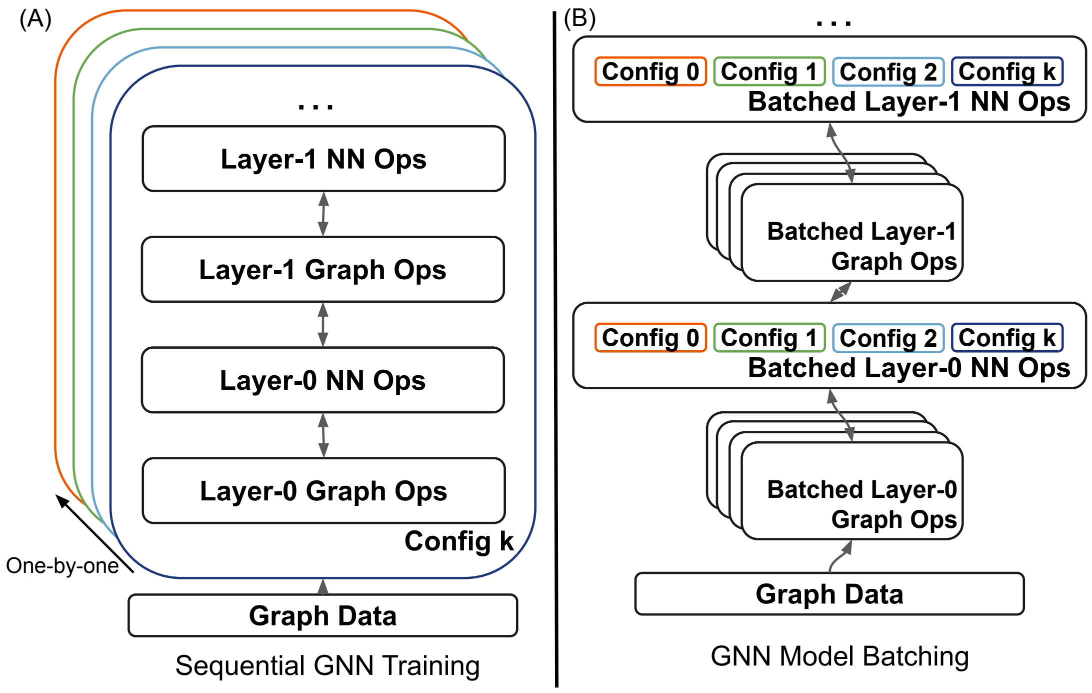


Figure 5.8. (A) Sequential training (B) Model Batching.

All data transmitted between the Graph Engine and DL Engine are also batched together. The models can then share all the data access operations to amortize costs.

5.6 Analysis of Cost Models

To better understand the problem, we model the various costs of GNN training: replication, computational, memory, and overheads. As mentioned earlier in Section 5.4.2, these models are not used in the Planner and only for deeper understanding and further experiment evaluation. Due to space constraints, we leave the tedious details and equations to C. We present a summary of two key observations about our cost model that we will validate empirically later.

Effect of Number of Partitions. The number of data partitions interplays with system performance in two ways: First, for large-scale dataset, more partitions are required to reduce memory pressure. Second, increasing the number of partitions will also increase the degree of parallelism and utilization because our Graph Engine uses one thread per partition.

To put it into an equation. We have the total computational cost for an execution plan with partitions:

$$W_P = \frac{W}{P} \max(\frac{P}{ML}, 1) + f_{overhead}(\frac{P}{ML}), \tag{5.3}$$

where W is the total amount of work (unit: time), $\frac{W}{P}$ is each partition's amount of work, $\frac{P}{M}$ is the total amount of tasks each machine gets, $\max(\frac{P}{ML}, 1)$ is the total amount of rounds each machine executes. Without losing generality, assume $f_{overhead}$ follows a monotonic increase along with P. We can then reason that as the number of partitions P increases; the overall runtime would first decrease and then increase.

We see precisely this behavior in our tests. Due to space constraints, the experiment details are moved to C. Due to the intertwined effects of this one parameter, the runtime behavior becomes non-linear and difficult to capture with simple cost models. Instead, we use rule-based heuristics to tune the number of partitions: we set it to be the same as the total number of CPU cores of the entire cluster unless more partitions are required to alleviate the memory pressure. Fortunately, GNN workloads are highly predictable in runtime and resource consumption. If necessary, one can always do test runs (for 1 or 2 epochs of training is more than sufficient) to figure out the optimal config setting.

Effect of Model Batching. The intermediate embedding/gradient size of a GNN model greatly impacts runtime performance. Because of GNN Model Batching, Lotan can have inflated intermediates sizes. To be precise, the intermediate sizes will be multiplied by the model batching size. Consequently, for model batching, we expect to see a scaling up when increasing model batching size due to higher utilization until the returns diminish due to overheads. We run experiments and show the results in Section 5.7.2 and will see the expected behavior.

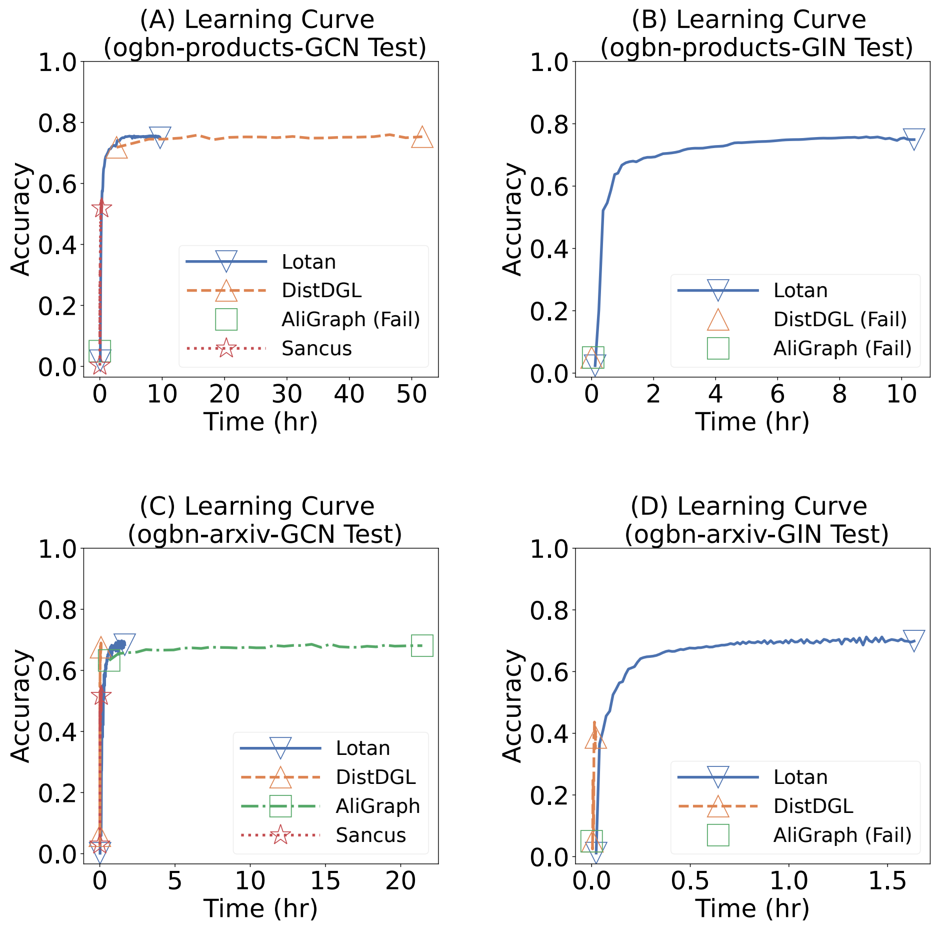


Figure 5.9. Learning curves for the chosen model on the test set. (A) ogbn-products-GCN. (B) ogbn-products-GIN. (C) ogbn-arxiv-GCN. (D) ogbn-arxiv-GIN. Corresponding learning curves on the validation set are presented in C.

5.7 Experiments and Evaluation

Prior Art. Out of the distributed GNN training systems discussed in Section 5.2.3, we show comparisons to the SOTA: DistDGL [323], AliGraph [326], and Sancus [230]. We excluded all systems that do not support distributed training and those without public release. Despite the best effort, we could not set up and use ROC [135], with a similar situation reported in [270]. Sancus [230] and PipeGCN [281] should be comparable systems, both with approximated processing, while others (including ours) are with exact processing, and we pick the former for benchmarking. Note that both DistDGL and AliGraph are primarily mini-batch GNN systems.

Although DistDGL can run full-batch training, it fails almost all our workloads. Therefore, we use it with the mini-batch setting. Mini-batch training is mathematically different from full-batch training. But we put in our best effort for a fair comparison by tuning the mini-batch size to their advantage and using standard benchmark metrics agnostic of the training scheme.

Datasets. We use three of the standard benchmarking datasets from OGB [124], which has become the go-to place for graph datasets for benchmarking. We use ogbn-products, ogbn-arxiv, and ogbn-papers100M. Additionally, we also include datasets reddit [113] and amazon [118], the original amazon dataset is not shipped in graph form, and we converted it to graph after acquiring a recipe from the authors of [135, 270]. The prior art also commonly use these datasets in their published papers. Table 5.2 first column shows brief statistics about the datasets.

Workloads. We define a GNN training workload with hyperparameter tuning factored in, which is an inevitable part of the end-to-end development of a GNN model. We primarily focus on two model architectures: GCN [150] and GIN [302] with various hyperparameter configurations. DistDGL and AliGraph have the batch size to tune additionally. In the corresponding literature, we found a batch size from 128 to 8192 is common. We tried as much as possible to make the comparison apples-to-apples and tune the batch size beforehand for them. To not understate their performance, we set the batch size to be as large as they could handle before failing to enable the maximum possible throughput. This means mini-batch size 8 for DistDGL on Amazon, 128 for DistDGL on ogbn-products+GCN and 8192 on ogbn-arxiv+GIN. And mini-batch size 128 for AliGraph on ogbn-arxiv+GCN. For Sancus, we can only test it on the GCN workloads as it does not have an existing implementation for GIN.

Experiment Setup. We use one cluster on CloudLab [246] with 8 worker nodes. Each node has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, and 10 Gbps network. Each worker node also has an Nvidia P100 GPU, which has 12 GB memory. We tried to get GPUs with larger memory but such resources are scarce and costly to obtain, especially for the long-running tests we do. Nevertheless, it is not a showstopper as Lotan's scalability gain is agnostic to the underlying hardware. Furthermore, even with larger GPUs, workloads can still scale beyond GPU memory capacity and would not change our observations about Lotan's scalability. All nodes run Ubuntu 20.04. We use Spark 3.2.0, Pytorch 1.10, and CUDA 11.0.

5.7.1 End-to-end Performance Study

We use a 3-layer GCN with a hidden layer size of 256, as described in [124], dubbed GCN. We also include a variant of it with hidden size 512, which we call GCN-Large, to further distinguish between Lotan and Sancus. We skipped DistDGL and AliGraph with GCN-Large due to their crashes or much longer runtimes, and these tests would not provide extra insights. For GIN, we use one from [302] that is 4-layer. For the MLPs in GIN, we use a 2-layer MLP with dimensions $\{128, 256\}$ for the end-to-end study. For the GCNs, their ApplyVertex functions are single-layer perceptions, while the GIN model uses the MLP described above. All of these models do not employ an ApplyEdge function and use a summation as the Aggregation.

Following the standard practices [150, 124, 302], we use an early termination of 10 epochs based on the validation set; we terminate if the validation accuracy does not increase for 10 consecutive epochs (with a tolerance of 0.01%). Further, we put a hard time limit of 48 hrs for each model config. We also combine the hyper-parameters used in the papers above to form a grid search: learning rate in $\{0.05, 0.01\}$, optimizer in $\{Adam, Adagrad\}$, and dropout in $\{0, 0.5\}$.

Table 5.2 summarizes the results of our end-to-end tests. On the ogbn-products + GCN workload, Lotan achieves 47x higher throughput than DistDGL while providing the same level of accuracy. There is no consensus from the GNN model research community on whether full-batch or mini-batch training is superior. Further, note some of the models and systems adopted mini-batch training partly due to the scalability issues of full-batch training [113, 67]. Lotan is designed to mitigate the said issues. Our finding of full-batch training achieving the same or slightly higher accuracy than mini-batch training is in line with prior work [135, 281]. Sancus, though it runs fast, has severe issues in accuracy during our test, likely due to its approximate nature. Furthermore, it starts to fail on the GCN-Large workload, but Lotan can still scale. Lotan is also the only system to be able to handle GIN training, and all other systems fail due to GPU memory issues. Increasing GPU memory might fix their problems on these specific workloads. Still, it would not resolve the fundamental issues these systems have and would not change the argument that Lotan has better scalability to handle large workloads.

On the tiny ogbn-arxiv dataset, while Lotan can still provide the highest accuracy on both GNNs, it no longer offers higher throughput than DistDGL. On the reddit dataset, which, despite having a similar number of nodes to ogbn-arxiv, has more edges, both DistDGL and AliGraph fail, likely due to the density of the graph. Sancus is still capable of operating and appears not affected much, but as other experiments showed, it offers lower accuracy due to its approximate nature. On the amazon dataset, Lotan and DistDGL are the only systems able to run the GCN workloads, and only Lotan for the GIN workloads. Lotan can provide a higher throughput than DistDGL. On the ogbn-papers100M dataset, one of the largest benchmark datasets available, Lotan is the only system able to run the workload. As far as we know, this is the first time for a GNN system to demonstrate full-graph GCN with a hidden size as large as 256 on this dataset. However, the execution is heavily bottlenecked, and a huge amount of disk spills happen. Consequently, we could not run the workload to converge in any reasonable amount of time. We only report the throughput numbers.

Figure 5.9 shows the learning curves for the best model out of some of the hyperparameter tuning workloads. On all workloads, Lotan converges fast and reaches the same level of accuracy as the SOTA. Regarding resource utilization, Lotan has high CPU utilization but generally lower GPU utilization across the workloads. This is because Lotan puts neural network operations on GPU and graph operations on CPU, while other systems put both on GPU. Except for Lotan, all other systems showed little to no disk R/W because they are not secondary-storage-aware, whereas Lotan can utilize the disk for spilling. Sancus and DistDGL further utilize GPU for communications, resulting in seemingly higher GPU utilization.

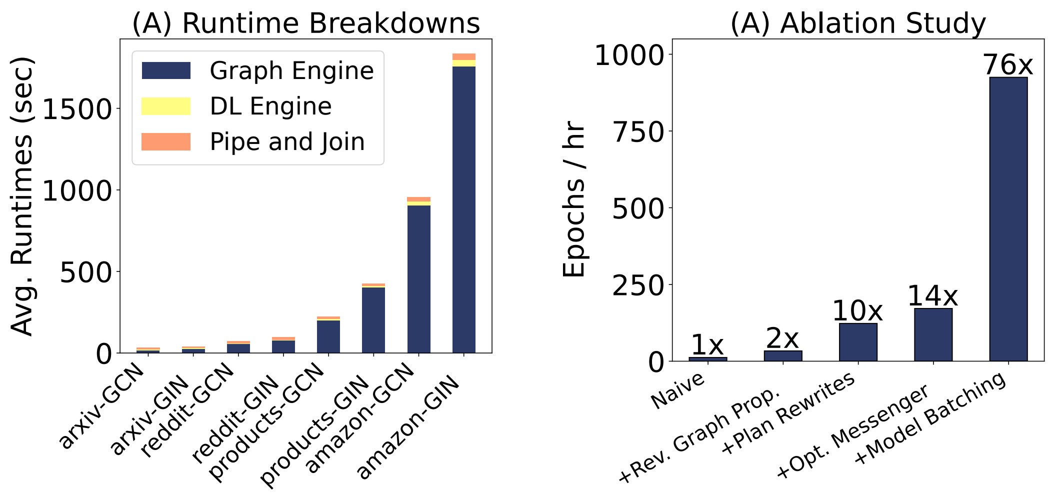


Figure 5.10. (A) Runtime breakdowns. (B) Ablation study.

5.7.2 Drill-down Experiments

To dig into the runtime figures, we also break down Lotan's runtime and investigate each portion's time costs in Figure 5.10(A). The Graph Engine costs dominate, especially on the larger dataset. DL Engine and Pipe-Join costs are not as significant. This composition will change when we try scaling the model in Section 5.7.2.

Ablation Study

To inspect each component's performance, we conduct an ablation study where we add our innovations to a naively implemented version of Lotan. We pick the ogbn-arxiv+GCN workload for this test. Figure 5.10(B) shows the results. We separate our technical innovations into four modules: (1) Reverse Graph Backprop (RGB) and the coupled GNN-centric partitioning scheme, as described in Section 5.5.1. (2) The execution plan rewrites by our Planner outlined in Section 5.4.2. (3) The various efforts we put into optimizing our Messenger architecture as described in Section 5.4.3. (4) Finally, our GNN Model Batching scheme proposed in Section 5.5.2.

All of the components have substantial contributions to performance gains; Reverse Graph Backprop can boost the performance by 2x without any plan rewrites or other optimizations. With Planner rewrites introduced, we get another 5x speed-up due to the sheer amount of communication and computation saved. Furthermore, our Messenger optimizations boost the performance by another 40% by reducing overheads in I/O, IPC, and synchronization. Last but not least, GNN Model Batching contributes a more than 5x speed-up due to amortized graph data access overheads. Overall, our technical innovations can boost the throughput of GNN model training by 76x, compared to a naively implemented system.

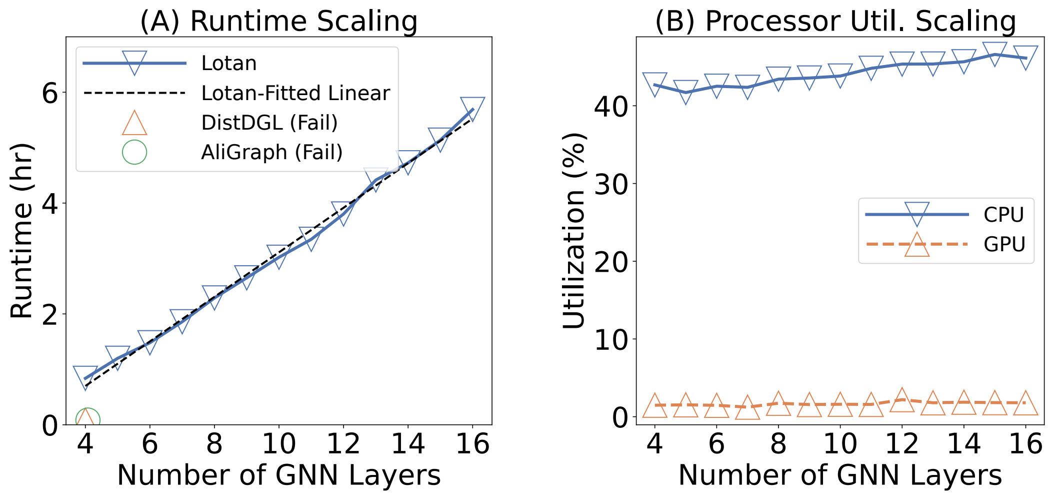


Figure 5.11. Depth Scaling. (A) Runtime. (B) Utilization.

Model Scalability

We now test Lotan's capability of scaling to larger neural network models. In practice, there are two primary ways to scale up a model: make it deeper by adding more (GNN) layers, or increase the number of neurons in each layer. We call the first type depth scaling and the latter type width scaling. Since Lotan disaggregates the graph operations from neural network operations, it has very different behavior for the two types of scaling. To thoroughly test it, we use two different workloads based on the GIN model used earlier and train them on the ogbn-products dataset. For the depth scaling test, we test with different numbers of GNN layers

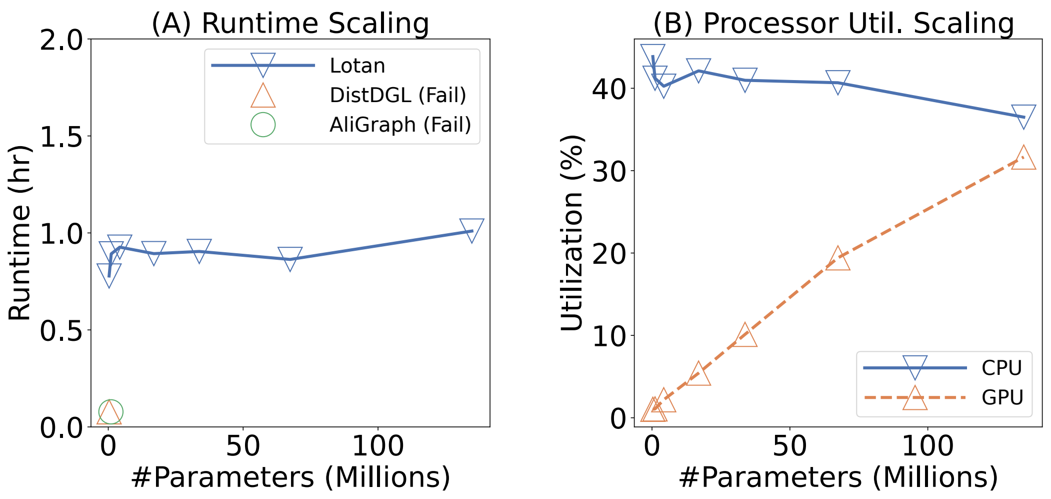


Figure 5.12. Width Scaling. (A) Runtime. (B) Utilization.

ranging from 4 to 16. For the width scaling test, we fixed the model to be a 4-layer, and we varied the size of the MLP in GIN from 128 to 2^{17} (131072); we kept the embedding size also fixed as 256. This results in various models with hugely different sizes.

Depth Scaling. Figure 5.11 shows the results: Lotan can easily achieve almost linear scaling to 16 layers and even beyond, and there is minimal fluctuation in the processor utilizations. Note that the scaling is linear but not proportional; when the number of layers doubles, the runtime does not; this is because the scaling follows $y = kx + b$ with a non-zero intercept. It is to be expected as the amount of work grows linearly in this case, and Lotan shows resilience at scale. To the best of our knowledge, Lotan is the first system to demonstrate scale to 10+ layer GNNs with full batch training. It is important to note that the systems we compared all failed at 4 or more layers, as already discussed in Section 5.7.1.

Width Scaling. We show the width scaling results in Figure 5.12. Increasing the MLP size will not increase the amount of work on the Graph Engine, and since the GPU was under-utilized when the NN is small, we see an almost constant scaling of Lotan. In this case, we see a dramatic increase in GPU utilization and almost constant CPU utilization. Thanks to the decoupling of graph and neural networks, scaling one side does not necessarily affect the other.

Lotan can provide independent scaling and frees the user from scalability issues. It enables the user to design the GNN components separately and more freely. Furthermore, Lotan can gracefully handle a GNN model with 140M+ parameters with full batch training. To put it into perspective, this is the number of parameters of some early Transformer DL models have: BERT (110M) [81], and GPT-1 (117M) [236]. To our knowledge, Lotan is the first system to be able to handle this scale among the GNN systems. As shown in Section 5.7.1, other systems all failed at the very beginning.

Model Batching

We now inspect the effect of model batching on the workloads. For this test, we take the same ogbn-arxiv+GCN models used in Section 5.7.1 and create workloads with various degrees of model batching. Figure 5.13 shows the results. We first notice that the time costs scaling is all linear with constant overheads (manifested as the intercept), per our cost model described in Section 5.6. There is also a substantial gain in throughput, especially at the low degree of the model batching regime. The SGC and AAA costs scale far less steeply than the SGC costs; therefore, as the model batching size increases, the SGC costs become more and more dominant. This indicates that the biggest challenge is on graph data processing.

At a low degree of model batching (< 10), the time costs are dominated by their respective constant parts and not scaling as much with the model batch size. Therefore, the time costs only increase around 3x while the model batching size rises from 1 to 10, resulting in throughput gains. However, as the degree of model batching increases, the scaling parts of time costs dominate, and we see 2x increase in time costs when batch size increases from 10 to 20 (2x increase). Consequently, the throughput scaling plateaus out as 2x model batched would mean 2x more runtime in this realm.

5.8 Conclusion and Discussion

Conclusions and limitations. By carefully abstracting, optimizing, and testing, we have demonstrated that it is possible to bridge the gap between graph analytics systems and DL systems with high scalability and without modifying their internal code. Currently, Lotan has two major limitations: (1) Lotan currently is only optimized for full-batch training. Mini-batch training would require efficient graph sampling and filtering, posing another scalability challenge and potential query optimization questions. (2) Lotan is meant for large workloads with large graphs and/or models. There is still some room for improvement on smaller workloads that do fit in memory, where more leeway for caching and batching techniques exists.

Discussion. Our results showed that the graph data system bottlenecked many of our tests (see Figure 5.10(A)). There are many sync barriers, costly data replications, and frequent garbage collections. Graph data systems need to evolve to better support GNN workloads and property-rich graphs with high dimensional dense vectors. Furthermore, GNN systems such as Lotan can be extended to adopt recent advances in ML systems research for optimizing model selection workloads [214, 319, 161, 213, 178], fine-tuning and transfer learning workloads [209], and large model scaling [216, 207]. It is non-trivial to extend their techniques designed for IID data to graph data. However, with suitable adaptation, they could further amortize some runtime overheads to make GNNs more efficient at scale.

Chapter 5 contains material from “Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines” by Yuhao Zhang and Arun Kumar, which appears in Proceedings of VLDB Endowment Volume 16, Issue 11, August 2023. The dissertation author was the primary investigator and author of this paper. All of our source code, data, and other artifacts are available at https://github.com/makemebitter/lotan.

Table 5.2. End-to-end test results. TLE: time limit exceeded (48 hrs per model). *For these tests, all models within the workload learned too slowly and got terminated too soon. To not understate the system's capability, we make an exception and include a separate run for a fixed 500 epochs and then pick the best valid accuracy checkpoint. †These tests would take an unreasonable amount of time to finish. Therefore, we did not train them to converge and only reported the throughput numbers.

Dataset Summary	Dataset	Model	System	Test Acc. (%)	Runtime (hr)	Throughput (epoch/hr)	CPU Util. (%)	GPU Util. (%)	Disk R/W (GB/hr)	Network (GB/hr)
#Nodes: 169.3K #Edges: 1.1M Avg. Degree: 13.7	arxiv	GCN	Lotan	**69.28**	1.90	924.85	18.47	5.79	4294.66	4203.44
			DistDGL	68.49	0.33	1912.50	21.39	15.80	3.45	5959.28
			AliGraph	68.60	171.44	1.59	5.35	6.73	3.48	44.46
			*Sancus	*55.23	*0.79	1855.67	5.98	89.97	3.00	19706.78
		GIN	Lotan	**71.22**	3.71	557.38	18.11	5.93	3123.74	4959.36
			DistDGL	43.64	0.15	1035.97	20.13	16.65	4.13	6109.54
			*DistDGL	*69.26	*2.76	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 232.9K #Edges: 114.6M Avg. Degree: 492.9	reddit	GCN	Lotan	**94.50**	16.81	77.82	30.80	0.85	6173.45	4154.67
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
			Sancus	92.67	0.04	1408.69	5.89	75.38	113.57	14160.26
		GIN	Lotan	**94.91**	23.15	50.16	30.47	0.83	6366.86	4083.77
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 2.4M #Edges: 61.8M Avg. Degree: 50.5	products	GCN	Lotan	**75.59**	63.82	16.22	49.58	1.75	6779.44	4989.52
			DistDGL	75.32	365.53	0.34	8.01	27.94	3.6	4548.72
			AliGraph	TLE	-	-	-	-	-	-
			Sancus	54.76	1.84	350.83	6.33	89.50	3.31	23086.58
		GCN-Large	Lotan	**75.89**	178.11	6.41	48.77	1.54	5917.79	3876.79
			Sancus	Fail	-	-	-	-	-	-
		GIN	Lotan	**75.75**	104.68	9.43	47.30	1.89	6735.52	4832.47
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 8.6M #Edges: 243.9M Avg. Degree: 28.2	amazon	GCN	Lotan	82.22	50.99	4.56	44.16	1.18	5451.33	3428.30
			DistDGL	**86.14**	261.58	0.02	10.66	51.19	4.48	4863.49
			AliGraph	Fail	-	-	-	-	-	-
			Sancus	Fail	-	-	-	-	-	-
		GIN	Lotan	**91.79**	252.84	2.26	39.37	1.08	4934.63	2230.87
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
#Nodes: 111.1M #Edges: 1.6B Avg. Degree: 29.1	papers100M	GCN	†Lotan	†-	†-	0.08	25.03	0.22	2499.20	801.93
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-
			Sancus	Fail	-	-	-	-	-	-
		GIN	†Lotan	†-	†-	0.04	24.16	0.11	2530.61	811.36
			DistDGL	Fail	-	-	-	-	-	-
			AliGraph	Fail	-	-	-	-	-	-

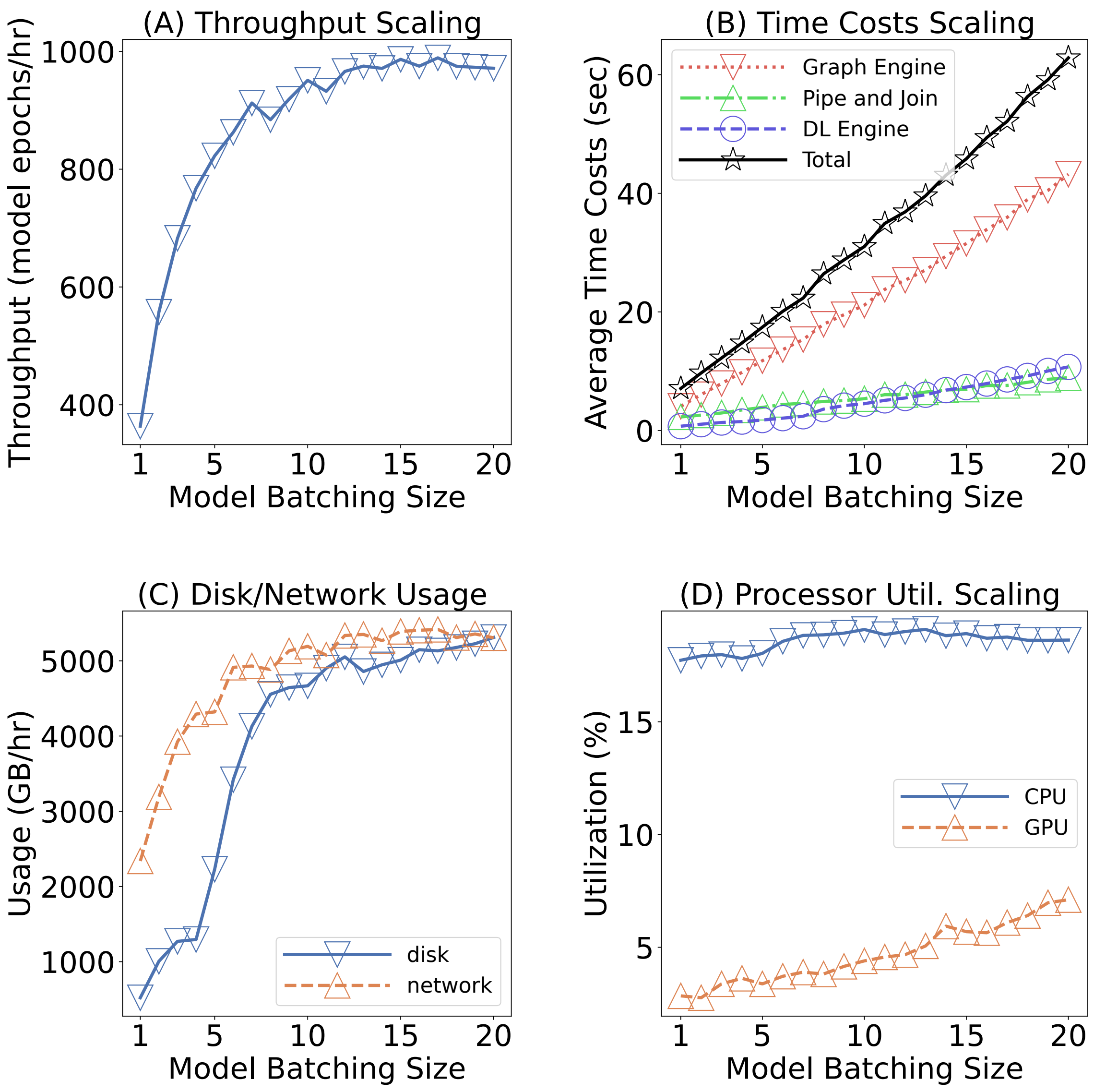


Figure 5.13. Scaling with GNN Model Batching. (A) Throughput. (B) Time Costs. (C) Disk and Network Usage. (D) Utilization.

Chapter 6

PANORAMA: Multimedia DB-style Retrieval with DL Inference

6.1 Introduction

In this chapter, we move on to model inference issues on large volumes of unstructured data, particularly videos. Videos are a ubiquitous and growing fraction of real-world data. For instance, YouTube alone gets hundreds of Petabytes of videos each year [53]. Thus, real-time *video monitoring* applications involving automatic recognition of objects in videos are gaining importance in many domains, including surveillance security [3], crowd control [4], traffic analytics, species monitoring, and more. The state-of-the-art approach for visual recognition is to use deep convolutional neural networks (CNNs) [108, 168]. However, deep CNNs are compute-intensive and have high inference latency, e.g., the popular Mask-RCNN [116] takes 1s for just five frames. Thus, enabling efficient visual recognition queries over video streams is a pressing data systems challenge.

Several recent lines of work in the multimedia, database, and systems communities aim to build more efficient systems for real-time video querying [311, 142, 282, 123, 143]. A common theme is to reduce CNN inference latency with cheaper models with smaller prediction *vocabularies* and using "cascades" of classifiers. But from conversations with video monitoring application users across domains such as surveillance and species monitoring, we find a pressing gap in the existing landscape of systems: *unbounded vocabulary*.

Problem: Unbounded Vocabulary. Almost all popular object recognition CNNs today handle only a finite "closed world" vocabulary of known targets. This is a natural consequence of their training dataset, typically a benchmark dataset like ImageNet [248], Pascal VOC [85], or MS COCO [185]. For instance, Pascal VOC has a tiny vocabulary of only 20 classes, e.g., "person," "bird," and "car." So, models trained on it only tell apart these 20 classes. This may suffice for some applications that only need to tell apart these classes (e.g., for self-driving cars), but for many emerging video monitoring applications, the query requirements are *more granular*: "Who is this person?," "What car model is this?," "What bird species is this?," and so on. In these applications, the target class set is not universally fixed and finite but rather growing over time, sometimes rapidly. For instance, the set of all people or all car models evolves. We call such a prediction target with a fast-evolving set of objects an *unbounded vocabulary*. Note the vocabulary needs to be the sub-classes of a common class, sometimes also known as "subclassing".

Example. Consider a CNN trained to tell apart dog breeds. Suppose its training dataset had a vocabulary of only three popular breeds: `Corgi`, `Labrador`, and `Poodle`. What will it output on an image of a rare dog breed, say, `Azawakh`? It will output junk probabilities for `Corgi`, `Labrador`, and `Poodle`. Of course, this not an issue with the model but rather its prediction vocabulary–a limited multi-class vocabulary is too restrictive. One might ask: Why not get labeled data for all possible classes? Apart from being impractical, such an approach also assumes new dog breeds will not arise. This is a fundamental issue for such applications: the prediction vocabulary is effectively unbounded. This issue is even starker for identifying faces in videos, e.g., for surveillance security, because it is impossible to get labels for all possible faces beforehand; furthermore, the set of faces is not bounded because new people will keep appearing.

Limitations of Existing Landscape. We see two main limitations. First, existing querying systems do not support unbounded vocabularies. Thus, their architectural assumptions and modeling choices need to be revisited. While the ML community has studied learning

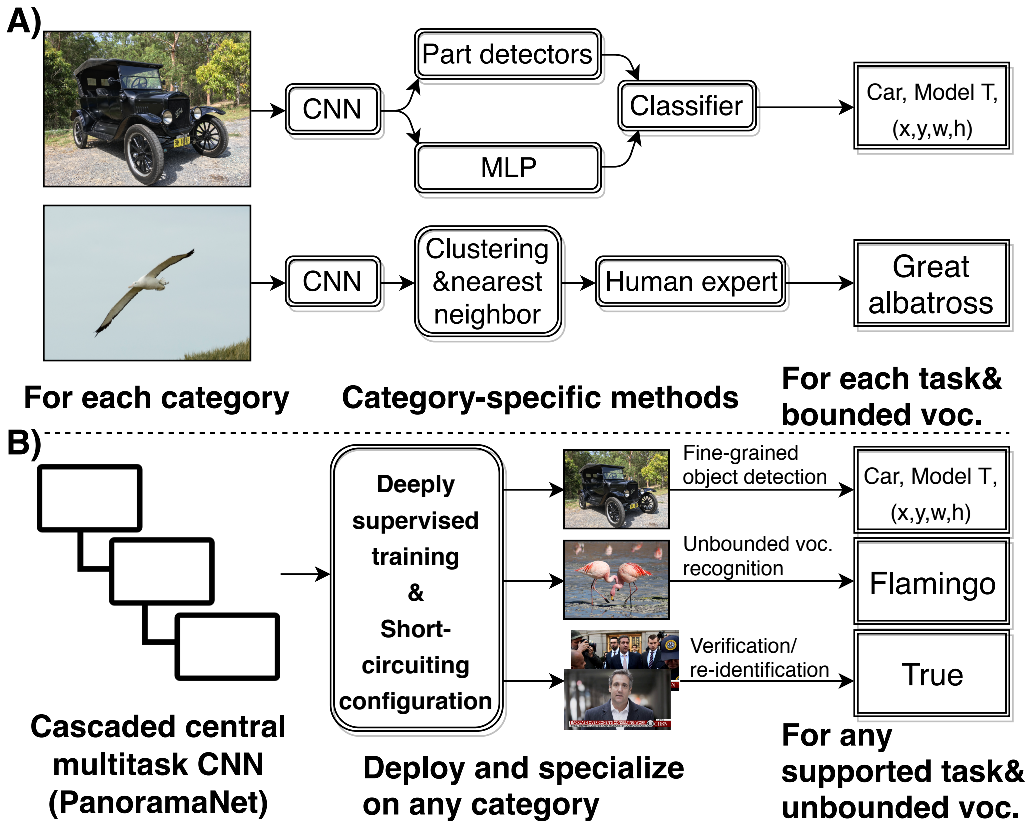

A)
CNN
Part detectors
MLP
Classifier
Car, Model T, (x,y,w,h)
CNN
Clustering &nearest neighbor
Human expert
Great albatross
For each category
Category-specific methods
For each task& bounded voc.
B)
Deeply supervised training & Short-circuiting configuration
Fine-grained object detection
Car, Model T, (x,y,w,h)
Unbounded voc. recognition
Flamingo
Verification/ re-identification
True
Cascaded central multitask CNN (PanoramaNet)
Deploy and specialize on any category
For any supported task& unbounded voc.

Figure 6.1. High-level qualitative comparison of existing vision stacks to Panorama's system design philosophy. (A) Each domain has a bespoke pipeline and a finite vocabulary. (B) Panorama enables unbounded vocabulary querying with a unified domain-agnostic data system that is automatically specialized for a given domain.

schemes to support new class labels, e.g., one-shot and zero-shot learning [167, 165], using them requires tedious manual intervention to re-train the model and provide metadata and/or more labels. This needs ML expertise, but video monitoring applications typically have only non-technical domain users in the loop of a deployed system (e.g., mall security). Second, making existing CNN-based stacks support unbounded vocabularies is not practical because they are often too application-specific and may involve bespoke pipelines, as illustrated by Figure 6.1(A). Such an ad hoc per-domain approach will duplicate the efforts of building, testing, and maintaining this capability across domains. Overall, the lack of support for unbounded vocabularies in a unified domain-agnostic data system is a bottleneck for emerging video applications.

System Desiderata. We have three related desiderata for a practical data system to support unbounded vocabulary queries over video. (1) Generalizing to new classes in the domain's vocabulary in an *automatic* manner without manual ML re-training. (2) Being *unified and domain-agnostic* to enable existing applications to adopt it without expensive manual customization. (3) Being *resource-efficient* and ideally real-time.

Our Proposed System. *We present Panorama, the first information system architecture for unbounded vocabulary queries over video.* It supports two kinds of queries popular in video monitoring. First is *recognition*: identify which *known object* (or set of objects) appear in a video feed (or an image database), e.g., a mall security officer checks a video feed against an image roster of wanted criminals to spot them in the crowd. Second is *verification*: tell if two frames (or images) have the same object in them regardless of whether the object is known, e.g., the officer compares an old frame with the current video feed to see if anyone reappeared. Our system design philosophy, illustrated by Figure 6.1(B), is to devise a unified and domain-agnostic system that can be automatically specialized for a given application.

Summary of Our Techniques. Panorama has three main components as Figure 6.1(B) shows: a new unified CNN architecture we call PanoramaNet, an automated offline training pipeline, and an online inference subsystem. PanoramaNet is a careful synthesis of three techniques (Section 4.1): multi-task learning from ML, embedding extraction from vision, and short-circuiting of inference from data systems. It helps meet desiderata (1) and (2). Our automated offline training pipeline is a synthesis of deep supervision (Section 4.2) and weak supervision (Section 4.3) ideas from ML. It helps meet desideratum (2). Finally, our online inference subsystem features a novel short-circuiting configuration technique (Section 4.4) that enables a tunable accuracy-efficiency tradeoff and a synthesis of nearest neighbor search from multimedia systems and query caching from databases to improve efficiency (Section 4.5). It helps meet desideratum (3).

Example Use-Cases. We present two example use-cases to highlight Panorama's functionalities. Section 3 presents the full query API of Panorama and a usage example. Note the user

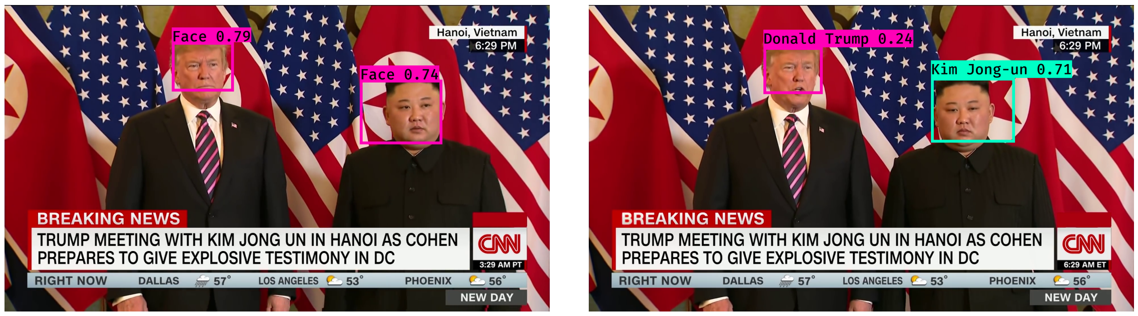


Figure 6.2. Example Panorama use-case (1): unbounded vocabulary recognition. Left: the frame shows two out-of-vocabulary faces (identities unknown) and the model only labels them as faces. Right: After the user freezes the video, clicks on the bounding boxes and labels them with names, Panorama can recognize these two objects in future frames. The scores shown on the left frame are the probabilities of being faces; on the right frame are the distances from the faces to their nearest neighbor in the *Known Objects* set.

interface is application-specific and orthogonal to our work, here we only show some possibilities of custom-built interfaces.

1. *Unbounded vocabulary recognition*. Figure 6.2 shows, say, a journalist spotting people in a news video feed. The vocabulary did *not* have Donald Trump or Kim Jong-Un to start with. Our system constantly detects faces and extracts embeddings from them. In this case, the journalist can select the bounding boxes and type in names for these two faces and their corresponding embeddings, respectively. Once it is done, these named embeddings are added into the known objects which serves as the vocabulary. From then on Panorama can recognize Donald Trump or Kim Jong-Un. This entire process happens on-the-fly and without any re-training of the CNN.

2. *Unbounded vocabulary embeddings*. Panorama can also output object embeddings (vectors) for further analyses. Figure 6.3 shows, say, a data scientist analyzing the representation of racial and gender groups in an Oscars video feed. The user runs an off-the-shelf clustering algorithm on the embeddings to get somewhat coherent clusters. Note that Panorama did not have any of these faces in its vocabulary.

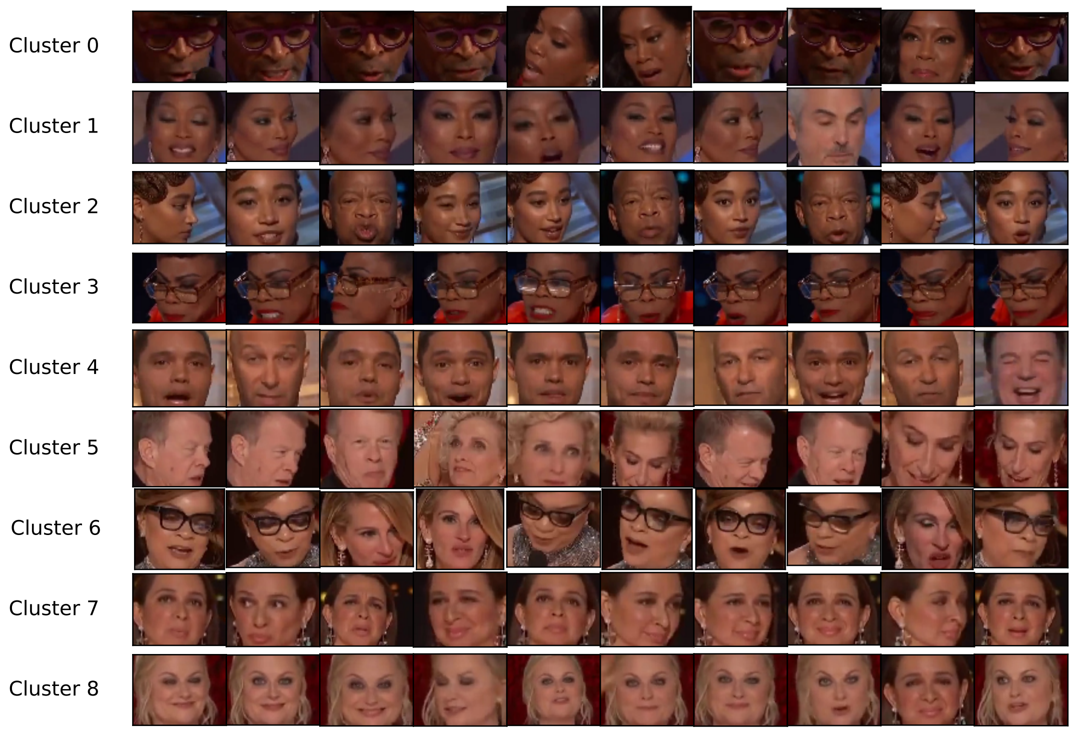


Figure 6.3. Example Panorama use-case (2): unbounded vocabulary embeddings extraction for faces. The embeddings are then clustered, yielding somewhat coherent clusters.

Our focus is on a new crucial system functionality for video monitoring applications: the deployed model need *not* be retrained when new classes (objects) arise. That is, users can just name the new objects from the video feed and add them to the known objects set–Panorama will *automatically* start recognizing them in the future. So, the users do not need any ML-related expertise or worry about retraining too often. The main technical novelty of this work is a new data system architecture that solves our real-world problem in a domain-agnostic, automated, and efficient manner. To achieve our goal, we draw techniques from diverse fields–vision, ML, databases, and multimedia systems–and synthesize and adapt them for our setting. We developed a new general CNN architecture, weak supervision scheme and auto-training scheme to enable such applications. We also studied trade-off spaces between accuracy and throughput. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first work to propose a unified information system architecture for unbounded vocabulary queries over video using CNNs.
- We create a new multi-task CNN architecture, PanoramaNet, that supports unbounded vocabularies in a unified and unsupervised manner based on embedding extraction and content-based image retrieval (CBIR). We present an automated and domain-agnostic training pipeline combining deep and weak supervision.
- We devise a novel self short-circuiting configuration scheme for PanoramaNet to enable practical accuracy-efficiency tradeoffs. We also create a query cache to improve efficiency further at scale.
- We present an extensive empirical evaluation of the accuracy, efficiency (throughput), and scalability of Panorama on multiple real-world videos. Overall, it offers between 2x and 20x higher throughput with competitive accuracy for in-vocabulary queries, while also generalizing well to out-of-vocabulary queries.

6.2 Setup and Background

We start by explaining our problem setup and defining some standard terminology from computer vision relevant for video monitoring applications. We then provide some technical background on multi-task deep learning and embedding extraction needed to understand our system.

6.2.1 Visual Querying Tasks

A video X is logically a sequence of image frames F_i, i.e., $X \equiv F_1 F_2 F_3 \dots$. This sequence can be unending for streaming video. The application specifies a *vocabulary* V of *objects* of interest it wants to identify in the images/video. The objects can be at any granularity, ranging from high-level generic categories (e.g., "person," "car," or "bird") to more fine-grained entity-level categories (e.g., "the person Donald Trump," "the car model Ford Mustang," or "the bird species California Quail"). Most prediction tasks in computer vision, as well as a suite of recent video querying systems, assume V is *finite*, perhaps even small. For example, NoScope [142] uses $|V| = 2$, viz., yes or no for a given object type like buses. In our setting, $|V|$ can potentially be infinite–we call this an *unbounded vocabulary*.

We are now ready to define the types of visual querying tasks of interest to us. We will then explain the implications of an unbounded vocabulary.

Definition 6.2.1 Recognition: *Given an image frame F, identify an object $v \in V$ present in F (if at all). The recognition is called* coarse-grained *if V only has generic object categories. It is called* fine-grained *if V contains entity-level categories too. A frame can have any number of objects. This task is also called* multi-object classification *in image-based applications.*

Definition 6.2.2 Localization*: Given an image frame F, identify the "regions" of F (e.g., bounding boxes) where all instances of objects from V are present (if at all).*

Definition 6.2.3 Verification*: Given two image frames F_1 and F_2, identify if the same "object" arises in both images; the object is assumed to be from V.*

The above tasks are not entirely orthogonal to each other. Real-world video frames often do not have only one object. Thus, localization is needed before or during recognition. The distinction between coarse- and fine-grained recognition is also not universal but rather application-defined; a fine-grained V typically distinguishes entities of the same type, e.g., identify the person instead of just is it a person. Fine-grained recognition often leads to unbounded V in real-world video monitoring applications, the focus of this work. For example, a recognition system may be trained on a finite set of people, but it should be able to recognize other people too during usage.

Recall that our goal is to enable unbounded vocabulary querying, both verification and recognition, for video monitoring applications. We now make a key observation that we exploit later in Section 3. If V is finite, verification can be mapped to two recognition queries and

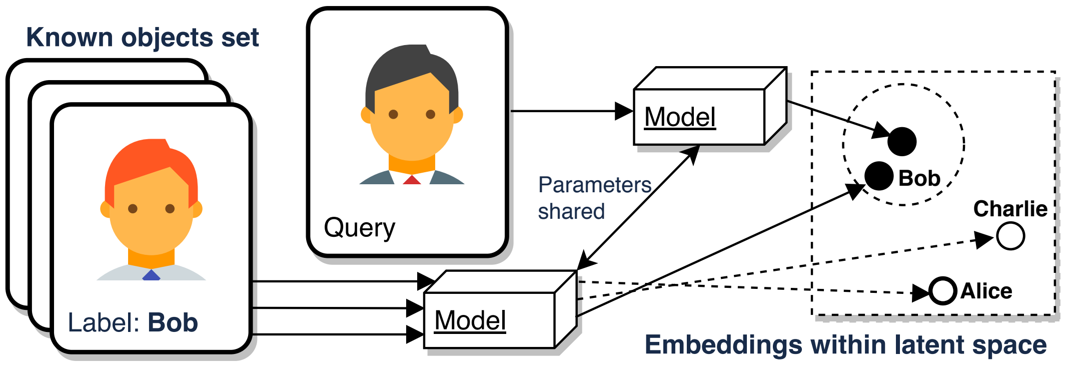


Figure 6.4. The embedding extracted from the query image is nearest to the known embedding for Bob in the metric space and farther from Charlie's or Alice's. This capability allows the model to distinguish between these entities.

comparing the labels. However, this is impossible for unbounded V. Instead, we reverse the mapping, since verification does not need to identify the label: cast recognition as multiple verification queries against known V.

6.2.2 Background: Multi-task Deep CNNs

Deep convolutional neural networks (CNNs) offer state-of-the-art accuracy for many computer vision tasks [168, 145] and have won many benchmark competitions [78, 248, 185, 85]. CNNs offer two critical ML engineering benefits [145]. First, they automatically learn salient features from the image during training instead of requiring extensive manual feature engineering [109]. This is done in the form of multiple layers of feature transformations involving operations such as convolutions, pooling, and non-linearity. Second, deep learning is highly flexible in terms of the structures of the inputs and outputs. In particular, *multi-task* deep learning can predict multiple related targets simultaneously while sharing most of the internal features for each task [239]. This capability is especially attractive for our problem since most of the processing for verification and recognition queries can be shared inside a single deep CNN. Later in Section 4.1, we explain how we leverage this capability in Panorama for unified processing.

6.2.3 Background: Embeddings

In both vision and language understanding, an embedding is an abstract representation of an entity in a metric space. Essentially, given a set of entities S, one learns a mapping $f : S \to \mathbb{R}^d$ that maps each entity to a d-dimensional vector. Embeddings are especially popular in deep learning since they enable almost all predictive processing computations to use only linear algebra operations. Embeddings have interesting semantic properties that allow us to tell apart entities. For example, FaceNet [254] can classify faces in a known set by extracting embeddings for each, while DeepFace [229] can extract such embeddings even without specific labels. In particular, one can often use distance measures in the high-dimensional space to distinguish between entities, as illustrated by Figure 6.4. This remarkable capability of embeddings has recently enabled more accurate CBIR applications [303, 316, 139]. Later in Section 4.1, we explain how we leverage this capability in Panorama to tackle the unbounded vocabulary problem.

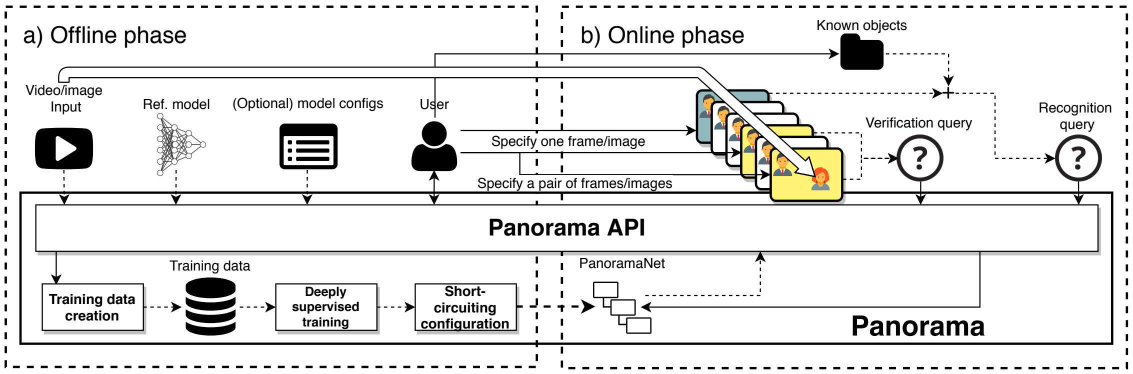


Figure 6.5. Overall system architecture of Panorama. Solid arrows represent invocations/interactions, while dashed arrows represent the flow of data/results. PanoramaNet is built once offline and then deployed for online video monitoring.

6.3 System Architecture and API

Overview. Recall that we have three main desiderata: support for unbounded vocabulary, automated domain-agnostic pipeline, and efficiency. To achieve all these, we design Panorama in a top-down manner with three main components, as shown in Figure 6.5. (1) A centralized *multi-task deep CNN* we call PanoramaNet whose parameters are automatically specialized for a given application, video feed, and a *reference* model; (2) An online phase to answer verification and recognition queries efficiently by short-circuiting PanoramaNet, also called *self-cascading*, possibly combined with *nearest neighbor search*; (3) A one-time offline process of automatic training data creation, training, and configuration of short-circuiting.

Queries and API. Panorama supports verification and recognition queries (Section 2.1). It also supports variable numbers of objects per frame, since it also performs localization implicitly. Table 6.1 lists the functions in our API, and Listing 1 gives an end-to-end usage example. The main querying routines are `verify` and `recognize`. The `album` is a set of known object images to recognize the video stream, e.g., known people or car models. Panorama allows this set to grow *without* retraining–this supports an unbounded vocabulary, as was shown by the application in Figure 6.2. The `detect` routine is a fall back for bounded vocabulary recognition. The `embedding` routine extracts object embeddings from a frame; this was used for the application in Figure 6.3. The other routines are used for the offline phase, which we introduce next.

Table 6.1. Functions in Panorama API.

Methods	Action
`verify(frame_a, frame_b, target_acc)`	Verification
`recognize(frame, album, target_acc, cache)`	Unbounded-voc recognition
`detect(frame)`	Bounded-voc object detection
`embedding(frame)`	Embedding extraction
`data_gen(video)`	Data creation on the video feed
`fit(data)`	DSN training on the data
`qualify(data, delta, task)`	Configure short-circuiting

```
Parameters
----------
ref_model: the reference model required for model specialization and cascaded query processing
min_cluster_size: <optional> the minimum cluster size, as required by HBSCAN algorithm, only needed if the ref_model is
      embedding extractor
data_path: the directory to the dataset for model specialization
delta_i: <optional> the slack variable for the cascade intervals
task: the name of the task to qualify and
a_g: the target accuracy for query processing on verification tasks
```

```
album_path: the directory to the known objects set for recognition
cache: use the query cache or not for recognition

Examples
--------
>>> model=Panorama(ref_model, min_cluster_size)
# invoke data creation, model training and short-circuiting config
>>> model.data_gen(video_feed)
>>> model.fit(data_path)
>>> model.qualify(data_path, delta, task=["verification", "recognition"])
# run a verification query
>>> ver_result=model.verify(file://frame_1324, file://frame_3325, a_g=0.9)
# run a recognition query
>>> rec_result=model.recognize(file://frame_1324,album,a_g=0.9,cache=False)
```

Listing 6.1. Panorama API and example usage.

Offline Phase. The user provides a video/image feed, a relevant reference model, and optional configuration parameters for Panorama. The reference model solves the *bounded* vocabulary recognition task, e.g., identify a known set of faces. Panorama's goal is to mimic this model's accuracy on the known object set while generalizing beyond that set with higher efficiency. Using the reference model, Panorama automatically generates training data on the video feed (Section 4.3) and trains PanoramaNet (Section 4.2). If the reference model yields embeddings instead of labels, then a configuration parameter can ask Panorama to generate labels instead. The training of PanoramaNet implicitly configures its short-circuiting using a novel mechanism (Section 4.4).

Online Phase. The user specifies the `verify` and/or `recognize` query as explained above for monitoring the video. The user interface is application-specific and orthogonal to this work. The interface shown in Figure 6.2 is only an example. They also specify an *accuracy goal* (relative to the reference model) to customize Panorama's accuracy-efficiency tradeoff. Panorama extracts embeddings from the given frames and compares them for verification or recognition as appropriate. For recognition, a nearest neighbor search is performed during inference.

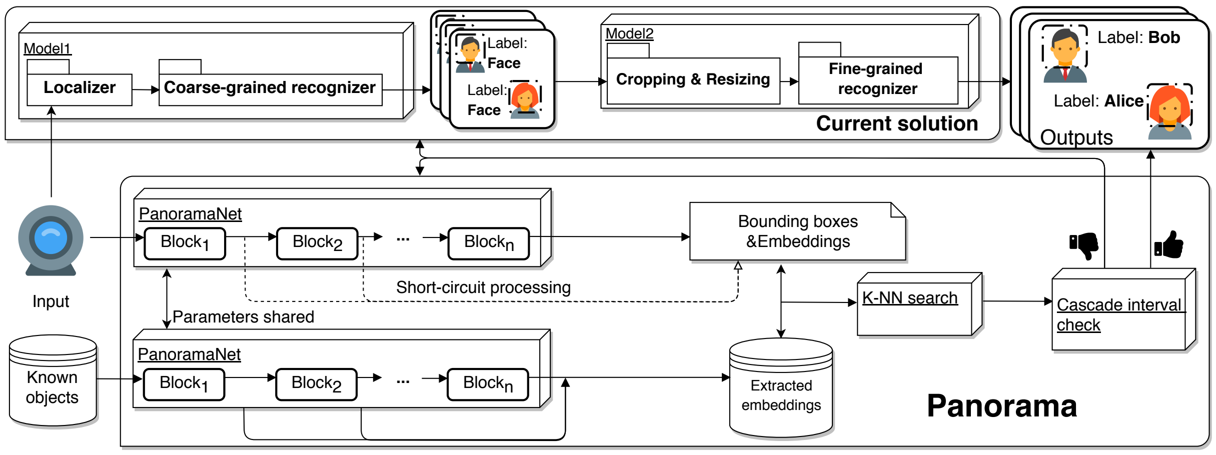


Figure 6.6. Detailed workflow of Panorama's internals for processing a recognition query. The deeply cascaded PanoramaNet can be short-circuited and is combined with nearest neighbor search for enabling unbounded vocabulary recognition in one pass. Also shown is a typical prior art solution; it takes a two-pass approach, with separate modules for coarse-grained and fine-grained recognition. The prior art solution also does not support unbounded vocabulary.

6.4 Components and Techniques

Most existing video querying models perform localization and recognition separately, and they do not support unbounded vocabulary. Adapting them to recognize new entities could require tedious manual retraining. In contrast, Panorama builds a single multi-task deep CNN that is automatically customized to each application. It "short-circuits" itself at query time to improve efficiency. Figure 6.6 illustrates Panorama's working in more detail. Next, we dive into each component: model architecture, training process, short-circuiting configuration, and inference process.

6.4.1 Deeply Cascaded Multi-task Model

Goals. At the heart of Panorama is a centralized multi-task deep CNN we call PanoramaNet. We have three goals for the design of this model. (1) Supporting both verification and recognition, as well as localization to identify multiple objects in a frame. (2) Being Domain-agnostic and not too tied to one application, e.g., faces or car models. (3) Being able to gracefully tradeoff inference cost for accuracy.

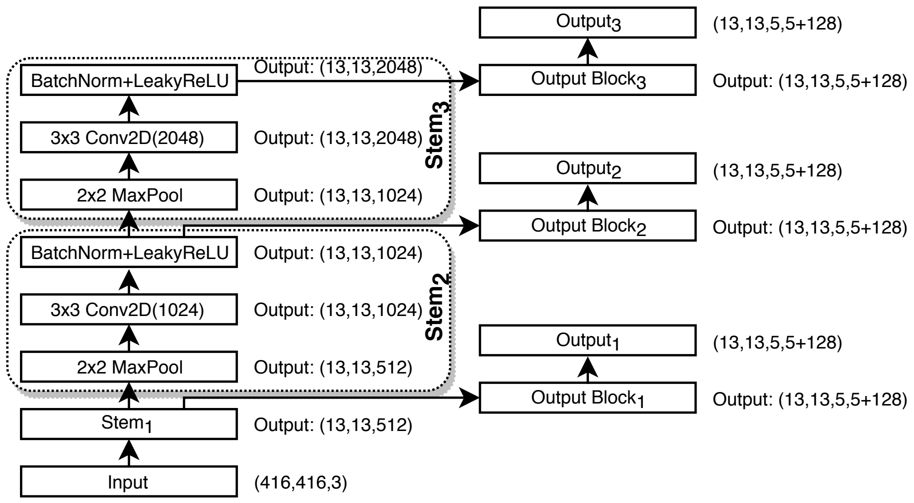


Figure 6.7. PanoramaNet deep cascade architecture with $n = 3$ blocks. An output has dimensions (grid, grid, number of bounding boxes, bounding box parameters+embedding dimension). All layers shown have a stride of 1 and are same-padded.

Basic Idea and Observations. To meet all three goals, we design PanoramaNet as a *multi-task* deep CNN with a *cascaded* modular structure. It has multiple trainable "blocks" of CNN layers, each with its own output block. The lowest layers of the CNNs act as a shared feature extractor for all blocks. All output layers have the same semantics, but they offer different accuracy-runtime tradeoffs. By short-circuiting at an earlier block, the inference cost goes down. Each output block has multiple intermediate targets for supervision during training, including bounding box regression, embedding extraction, and in-vocabulary recognition loss. This multi-task setup is what allows Panorama to simultaneously recognize in-vocabulary objects and generalize to out-of-vocabulary objects after deployment. During the training process, short-circuiting is configured based on a user-given *accuracy goal* and the model's accuracy on a validation set.

PanoramaNet Neural Architecture. Our model performs localization and embedding extraction jointly in one pass. Its base architecture is adapted from Yolov2-tiny [241] with two major modifications for our problem. First, while Yolov2 is a one-pass model for localization and recognition, it does not support unbounded vocabulary. Thus, we augment it by "wiring in" an embedding extraction module. Second, inside each grid cell that segregates the feature

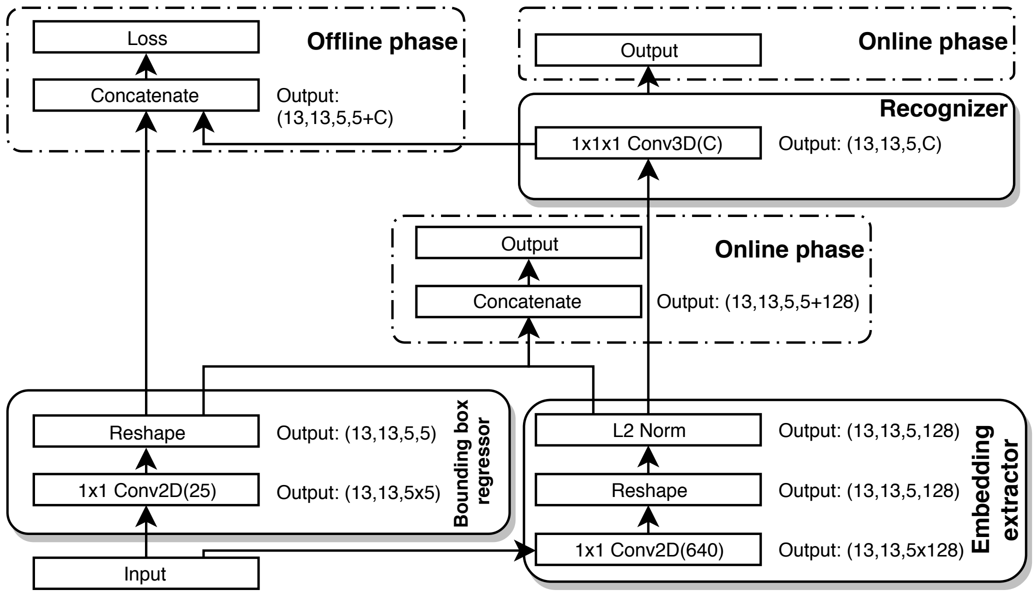
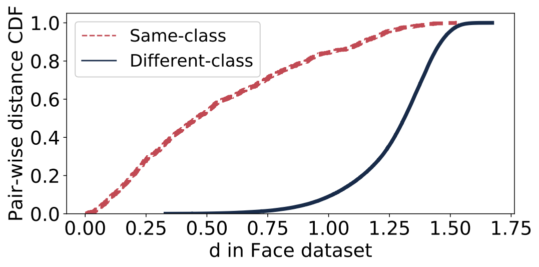


Figure 6.8. Architecture of Output Blocks from Figure 6.7. All layers are stride=1 and same-padded.

maps of the CNN layers in Yolo, the bounding box regressors (for localization) and recognizers work independently. So, even if the bounding box is poor, the recognizer may still yield correct labels. However, in our setting, this property is antithetical for embedding extraction, since the box-segmented image must align well with the object for embedding extractors to work properly. To tackle this issue, we confine each recognizer to its corresponding bounding box regressor via 3D convolutional layers.

Figure 6.7 shows the high-level architecture of PanoramaNet. Due to space constrains, Stem$_1$ architecture is presented in D. Figure 6.8 expands Output Block$_i$; layer are annotated with their size, name, and the number of filters, e.g., 3x3 Conv2D(1024) means a 3x3 2D convolutional layer with 1024 filters. PanoramaNet stacks many such Stem and Output Blocks. We collectively denote each Stem$_i$ along with its Output Block$_i$ as Block$_i$. We use an embedding dimension of 128. We use Euclidean distance (L2 norm) to compare embeddings.

Output Blocks. As Figure 6.8 shows, Output Blocks have modules that are used only during the offline phase. In particular, the embedding extractor is also trained during the offline phase using in-vocabulary labels. During the online phase, the recognizer module is applicable



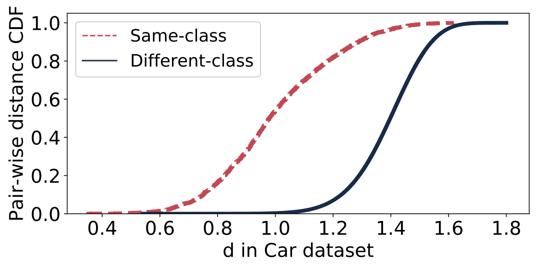


Figure 6.9. CDFs of pairwise Euclidean distances between the embeddings yielded by Block$_2$ of PanoramaNet.

only for in-vocabulary recognition, but the embedding extractor applies to both in- and out-of-vocabulary recognition. Due to our multi-task setup, all outputs have both bounding boxes and embeddings (or labels). Outputs are then thresholded based on the "objectness" of the bounding boxes (explained more in Section 4.2) and then thresholded with non-maximal suppression. Throughout the paper, we set the former to be 0.1 for verification and 0.03 for recognition, and the latter to be 0.5.

Answering Verification Queries

We now explain how PanoramaNet answers verification queries. The model outputs embeddings from each of the two input frames/images. We then simply *threshold* on the L2 distances of the embeddings as follows. Given two frames f_i and f_j and their corresponding embedding sets $\{e_i\}$ and $\{e_j\}$, the verification answer is *yes* if the following holds (otherwise, it is *no*):

$$\min_{i,j} ||e_i - e_j|| \leq \gamma,$$

In the above, γ is a Panorama configuration threshold to distinguish embeddings of different entities in the metric space. This approach works because as explained in Section 2.3, well-trained embeddings offer us this geometric capability to roughly tell apart different entity classes. But how to set γ? We set γ based on a held-out validation set during the offline training process. This requires a balancing act between precision and recall. To achieve this balance, consider the CDFs of the pairwise distances for same-class (e.g., same person) embeddings and different-class embeddings in Figure 6.9. On the *Faces* dataset (explained more in Section 5), a threshold of $\gamma = 0.8$ reasonably separates same-class pairs from different-class pairs with high precision. Similarly, on the *Cars* dataset, $\gamma = 1.1$ is suitable. We prefer such high-precision thresholds, since overall recall can be enhanced through other means, e.g., have multiple different images for known objects.

Answering Recognition Queries

As mentioned earlier, we map a recognition query to multiple verification queries. Given a query image's embeddings (e.g., from a video frame), we perform a nearest neighbor search against the embeddings in the album. This is done as bulk matrix arithmetic on the GPU, which turned out to be much faster than indexing. Thresholding can be used on top to ensure the retrieved neighbors are similar enough. Since recognition involves multiple queries, it is more prone to errors and harder to optimize. Thus, Panorama offers a configuration option of using the recognizer component in PanoramaNet for output labels directly for in-vocabulary recognition; note this is not possible for out-of-vocabulary recognition and only nearest neighbor search can be used.

6.4.2 Training with Deep Supervision

Since PanoramaNet has multiple output layers, we need to consider all of their loss functions during backpropagation. To this end, we use the "deep supervision" approach introduced in [169]. It was originally devised to tackle the vanishing gradients issue for accuracy and for better discriminative power of each layer. We repurpose it to enable our accuracy-throughput tradeoff; to the best of our knowledge, this is the first time deep supervision is exploited this way. The overall loss is as follows:

$$L = \sum \lambda_k l_k, \tag{6.1}$$

In the above, λ_k is the weight for output layer k and l_k is that layer's loss. Each l_k is backpropagated only through its parent layers. λ_k controls the trade-off between more opportunities of early-exit vs better over-all performance. We set each λ_k inversely proportional to the number of FLOPS to compute that layer's output. In particular, we set $(\lambda_1, \lambda_2, \lambda_3) = (8, 2, 1)$. We conduct experiments in Section 6.5 to study the effect of these weights. Note that our deeply cascaded architecture is generic; l_k can be any form of loss determined by the multi-task target goals. In particular, l_k in PanoramaNet is the same loss as in Yolov2; due to space constraints, we present the whole loss function in D.

Given Block$_k$, B is the number of anchor boxes, S is the number of grids, (x_i, y_i, w_i, h_i) are the location of the centroid, and the width and height of anchor boxes. C_i is the "objectness" of the output, referred to earlier in Section 4.1. λ_{coord} and λ_{noobj} are weights to balance the parts of the loss; we use the default weights from [242]. Finally, $p_i(c)$ is the classification "confidence" for class c. We adapt the code from [29] to implement our loss function.

6.4.3 Automated Training Data Creation

PanoramaNet is domain-agnostic and meets our systems-oriented goals. But it still needs to be trained on a specific application's data. To this end, we create an automated training process to customize PanoramaNet to a given data-set in the offline phase. We first run the user-given reference model on a portion of the video (or subset of images) to create "weakly supervised" training data [325]. The reference model must provide both bounding boxes and labels for the corresponding bounded vocabulary task. We also support models that produce embeddings instead of labels; in this case, Panorama clusters the embeddings and assigns a label per cluster. We use HDBSCAN [60] for clustering; the user can set its hyper-parameters during configuration or use defaults. We also denoise the clustered data by removing outliers. Overall, the reference model "teaches" Panorama, which means the reference model caps its in-vocabulary accuracy. If one desires higher accuracy, or if a reference model is not available for an application, the user has to give PanoramaNet a whole labeled dataset; we used this approach for the *Cars* dataset in Section 5.

6.4.4 Configuration of Short-Circuiting

Goals and Basic Idea. A critical design decision in PanoramaNet is its multi-block architecture, which enables a graceful accuracy-throughput tradeoff by short-circuiting. But when to short-circuit? Recall that the user sets an *accuracy goal*. We need to satisfy this goal reliably at query time. Our basic idea is to compute a "score" for a given query at each block and compare it against a pre-computed "cascade interval" for that block. If the query's score at a block falls in its cascade interval, it means the model is not confident about this intermediate output and so, subsequent blocks need to be invoked. Otherwise, we short-circuit at the current output and return immediately. We first explain how we use cascade intervals and then explain how we set them, including how our approach ensures correctness.

Using Cascade Intervals. We pre-compute a cascade interval $[L_i, H_i]$ for Block$_i$ in the offline phase. In the online phase, we are given a verification query with two frames/images f and g. Let d_i denote the distance between the pair of embeddings output by Block$_i$ for these frames; this is our score for short-circuiting. We start processing both frames from the first block until we hit a Block$_i$ such that $d_i \in [L_i, H_i]$. If no block satisfies this, we invoke the reference model, which acts as the "pseudo ground truth" in our weakly supervised setup.

Let a_g denote the user's accuracy goal. Let the actual accuracy of Block$_i$ be a_i on a given labeled set of examples $\mathbb{D}_v = \{((f,g),y)\}$, wherein y is the ground truth (yes or no); denote $|\mathbb{D}_v|$ by N. So, Block$_i$ has correctly answered Na_i queries. If $a_g > a_i$, it means Block$_i$ has a deficit of $N(a_g - a_i)$ queries to meet the accuracy goal. Thus, for short circuiting to succeed at Block$_i$, the percentage of queries that should have been answered correctly within the set of wrongly answered queries is $\frac{N(a_g-a_i)}{N(1-a_i)} = \frac{a_g-a_i}{1-a_i} \equiv q_i$ (say).

Setting Cascade Intervals. In the offline phase, we plot the CDF of d_i for queries that did *not* get correctly answered at Block$_i$ using the labeled validation set $\mathbb{D}_v$. We set $[L_i, H_i]$ to match the above percentage q_i of these queries. A natural choice is to select an interval around the median:

$$L_i = P(0.5 - \frac{a_g - a_i}{2(1-a_i)} - \delta_i, \mathbb{S}_e) \tag{6.2}$$

$$H_i = P(0.5 + \frac{a_g - a_i}{2(1-a_i)} + \delta_i, \mathbb{S}_e) \tag{6.3}$$

In the above, $P(x, \mathbb{S})$ denotes the x percentile of the set $\mathbb{S}$. $\mathbb{S}_e$ is the set of d_i for all examples in $\mathbb{D}_v$ such that short-circuiting at Block$_i$ gives the wrong prediction (i.e., the output is the opposite of y). Under the assumption that the validation set and deployment data come from the same or similar distribution, the above values guarantee that the accuracy goal will be met, while short-circuiting as much as possible. To account for statistical differences between the deployment data and validation set, we also include a small slack variable δ_i.

Correctness Analysis. We now explain why our above approach guarantees that the accuracy goal a_g will be met. Let the accuracy of Block$_i$ be $a_i < a_g$. Queries that fall into the interval $[L_i, H_i]$ at Block$_i$ all get sent to the next block. Note that since we do not have ground truth in the online phase, we do not know if Block$_i$ answered any queries correctly; we can only rely on c_i for short-circuiting. But note that exactly q_i fraction of all wrongly answered queries (and unknown numbers of correctly answered queries) are sent by Block$_i$ to a later block to be eventually answered correctly, perhaps ultimately by the reference model itself. Thus, the overall accuracy goes up from a_i to at least a_g by performing more inference computations (invoking more blocks) for queries that did not get short-circuited.

6.4.5 Query Cache

Intuition. Video with high frame rates lead to lots of queries, e.g., 40Hz means 40 queries for 1s. However, videos also have high *temporal redundancy*: most successive frames are similar. Thus, downsampling can raise efficiency without hurting accuracy much (e.g., 1 frame from 1s). But we can go further to exploit a key property of our target applications: objects typically do not appear and disappear too fast. Some objects may even last minutes, e.g., faces in news videos. This gives us to another systems insight: *cache recent query results to reduce computations for the same object*. Such a query cache skips the costly nearest neighbor search for successive recognition queries.

Mechanism. We create an approximate query cache with the embeddings since they exist in a metric space with Euclidean distance as an indicator of similarity. Denote $d(x,y)$ as the distance between embeddings x and y. Let e_a be the embedding from a recent frame. Let e' be the embedding of its nearest neighbor result from `known`. For a new frame with embedding e_b, we have one observation based on the triangle inequality; Suppose $d(e_a,e') \leq \gamma$, where γ is the threshold for same-class embeddings (Section 4.1.1). If $d(e_b,e_a) \leq d(e_a,e')$, return the label of e' as the result and skip the search. This approach is an approximation because a different embedding in the album may be nearer to e_b than e_a (although with low probability). Thus, our cache creates a runtime-accuracy tradeoff.

Corresponding to the observation, we cache the most recent several frames and evict in FIFO manner, the number of which is the cache size. Given a frame, we check the cache for hits and return the labels. We then take the misses and do a normal k-nn search to get labels for them. Finally we update the cache with all labels acquired in the above steps for this frame. Overall, this query cache can reduce runtimes significantly when the known objects set is massive.

6.4.6 Online Phase Inference Process

Figure 6.10 depicts how queries are processed in the online phase. For *verification*, both frames are passed to PanoramaNet for embedding extraction one block at a time. Pairwise distances between the embeddings are checked for short-circuiting. If short-circuiting succeeds, we threshold the distance against γ for the final answer (yes or no). For *recognition* queries, Panorama extracts embeddings from given frames and compares against the embeddings (for the corresponding block) in the known object set via a nearest neighbor search. This search might

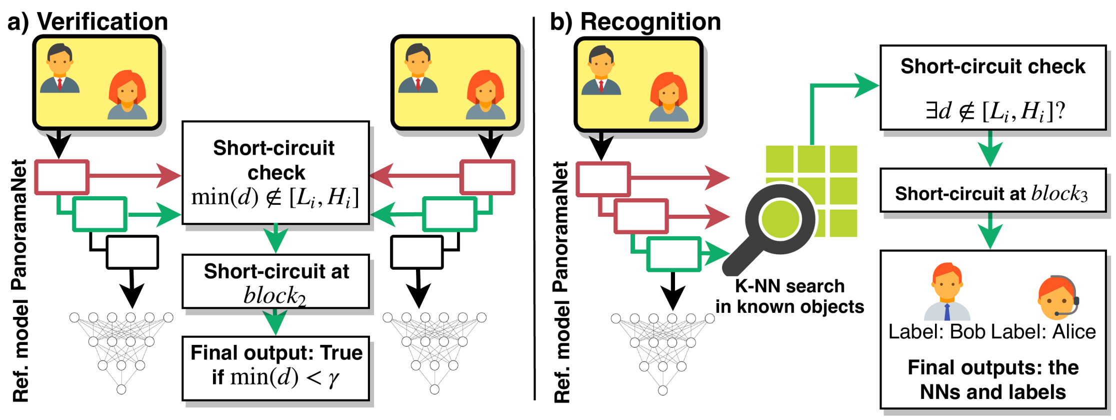

a) Verification
Ref. model PanoramaNet
Short-circuit check $\min(d) \notin [L_i, H_i]$
Short-circuit at $block_2$
Final output: True if $\min(d) < \gamma$
b) Recognition
Ref. model PanoramaNet
K-NN search in known objects
Short-circuit check $\exists d \notin [L_i, H_i]$?
Short-circuit at $block_3$
Label: Bob Label: Alice
Final outputs: the NNs and labels

Figure 6.10. Examples of Panorama's inference execution. a). The verification query is short-circuited at block$_2$. The left and right models including PanoramaNet and the reference model share parameters, respectively. b). The recognition query is short-circuited at block$_3$. Embeddings from the known objects were pre-extracted and stored.

potentially be skipped by the query cache (Section 4.5). Once again, if short-circuiting succeeds at some block, we stop and return the nearest result's label. As mentioned in section 4.4, the reference model is the fallback option in case none of the blocks of PanoramaNet can answer the query with high confidence.

6.5 Experiments

We now evaluate Panorama with several real-world workloads and datasets for both verification and recognition queries. In summary, our results show the following:

1. For in-vocabulary verification, Panorama offers between 2x and 8x higher throughput (lower latency) than a strong baseline, while offering competitive accuracy. For in-voc. recognition; the speedups are up to 20x.

2. Panorama generalizes well for out-of-vocabulary queries, offering much higher accuracy than random guessing baselines, while still offering high throughput.

3. Panorama configuration parameters enable a graceful tradeoff between accuracy and throughput.

4. As the known objects set size scales up for recognition, Panorama's query cache helps raise throughput up to 6x.

We first describe the datasets and workloads used. We then present the end-to-end performance results followed by a drill-down study of the contributions of Panorama's techniques. Finally, we present the scalability test.

6.5.1 Experimental Setup

Datasets. Table 6.2 lists our datasets. *Faces*[62] and *Birds*[163] are videos recorded from online surveillance cameras at 30Hz frame rate. *Faces* is for recognizing people; *Birds*, for recognizing bird species. All videos are decoded and unpacked into frames. We sample 1 frame per second. Our baseline models also operates on the downsampled frames instead of the original video, which makes them already strong baselines for the throughput-accuracy tradeoff we study. *Cars* is an image dataset for car model recognition [304].

Table 6.2. Datasets and reference models.

Dataset	Source	|Voc.|	#Frames	Ref. model
Faces	CBSN[62]	60	5.4m	MTCNN[313]+FaceNet[254]
Birds	Bird Cam[163]	6	5.4m	Yolo[241]+Inceptionv3[74]
Cars	CompCars[304]	431	45k	Yolo[241]+GoogLeNetCars[305]

Reference Models. Each reference model has two sub-models, as Table 6.2 shows. The reference model for *Faces* produces embeddings; thus, we create pseudo-labels after unsupervised clustering. Overall, the reference models operate on a bounded vocabulary. For *Faces* and *Birds*, Panorama is weakly supervised by the respective reference model (Section 4.3), but for *Cars*, we used the CompCars [304] dataset to show that Panorama can work on strongly supervised image data as well, not just week-supervised videos.

Data Split Methodology. Figure 6.11 shows how we split the datasets. We first split all examples into `train`, `val` and `test`. At the same time, we split the vocabulary into `in-voc` and out-voc. Then, test is further split into in-voc test, with test examples that have in-voc labels, and out-voc test, the rest of test. Only the in-voc train of train is used in the CNN training. val serves for the validation during training and short-circuiting configuration. Then at deploying time, we poll 5 best frames per class, based on Panorama's confidence score of object detection, from train and val to form known for subsequent recognition queries. Then known becomes the new vocabulary. For videos, we chunk the videos, instead of random order, into 60:20:20 ratio for the train-val-test split. The vocabulary is also chunked into 80:20 for in-out-voc split, sorted in descending order by the cardinality of each class. But for *Cars*, we reuse the pre-existing 70:30 train-test split in its original labeled dataset; however, its vocabulary is also split 80:20. The val split is 10% of train. Overall, PanoramaNet is trained only with in-voc train, which allows us to simulate the unbounded vocabulary scenario. At deployment time, we use known as the album for recognition queries.

Training PanoramaNet. We train PanoramaNet with Adam optimizer. Adam is configured with an initial learning rate of 0.5×10^{-4}, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$. We use a batch size of 64 for *Faces* and *Birds* and 8 for *Cars*. The training is terminated if for 10 consecutive epochs the validation loss does not improve. Training for face terminates after 48 epochs taking 2 day 15 hours. Training for bird terminates after 109 epochs taking 1 day 3 hours. Training for car terminates after 208 epochs taking 8 days 6 hours.

Accuracy Methodology. As explained in Section 6.4.3, we use the denoised outputs of the reference model as labeled data for Panorama. Thus, all in-voc test accuracy is reported relative to the reference model. This methodology is fair because our focus is not on improving absolute accuracy but rather systems issues of functionality and efficiency.

Evaluation Metrics. We have three main metrics: *throughput*, *verification accuracy*, and *recognition accuracy*. Throughput is the number of queries answered per second; it is based on the wall-clock time taken for all queries put together. Since we have no batch processing or task parallelism, the higher the throughput, the lower the query latency. We omit all frame

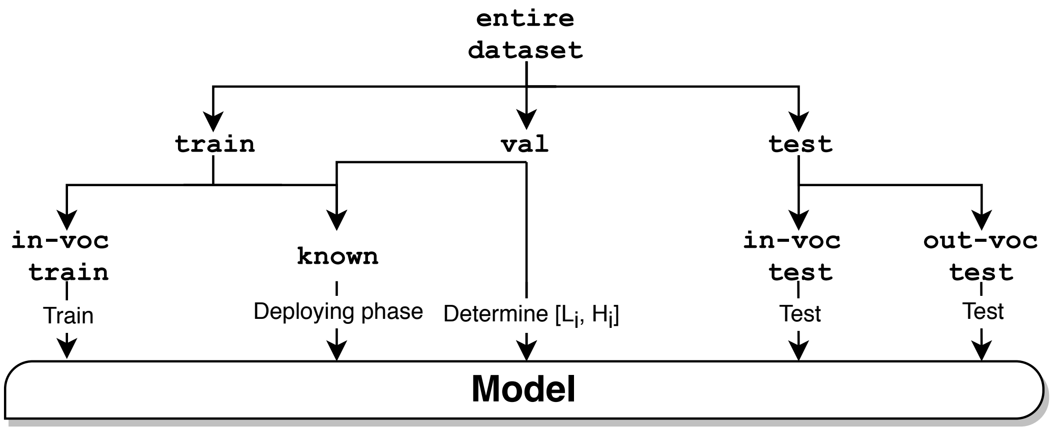


Figure 6.11. Schematic diagram about dataset split.

preprocessing time (e.g., decoding or resizing) for all compared approaches because they were minor.

Verification accuracy is defined based on standard practice [126] as the ratio of the number of queries that were correctly answered to the total number of queries. Given a verification query with a pair of frames/images (f, g), denote the sets of classes appearing in f and g by $\mathbb{O}_f$ and $\mathbb{O}_g$, respectively. The query with (f, g) returns yes if and only if $|\mathbb{O}_f \cap \mathbb{O}_g| \geq 1$; otherwise, the query returns no.

For recognition accuracy, we only evaluate it on frames on which the reference model gave output labels. On a frame containing l classes $\{Y_0, Y_1, ..., Y_i, ..., Y_l\}$, we ask the model to give at most m labels $\{z_0, z_1, ..., z_j..., z_m\}$. This gives us a standard metric called "top-m" accuracy; we set $m = 5$ for our experiments. Recognition accuracy at the frame level is now defined as follows.

$$k_r = 1 - \frac{1}{l} \sum_j \min_i \mathbb{1}_{z_i = Y_j}, \tag{6.4}$$

The overall recognition accuracy is then defined as follows, wherein $\mathbb{Q}$ is the set of all recognition queries:

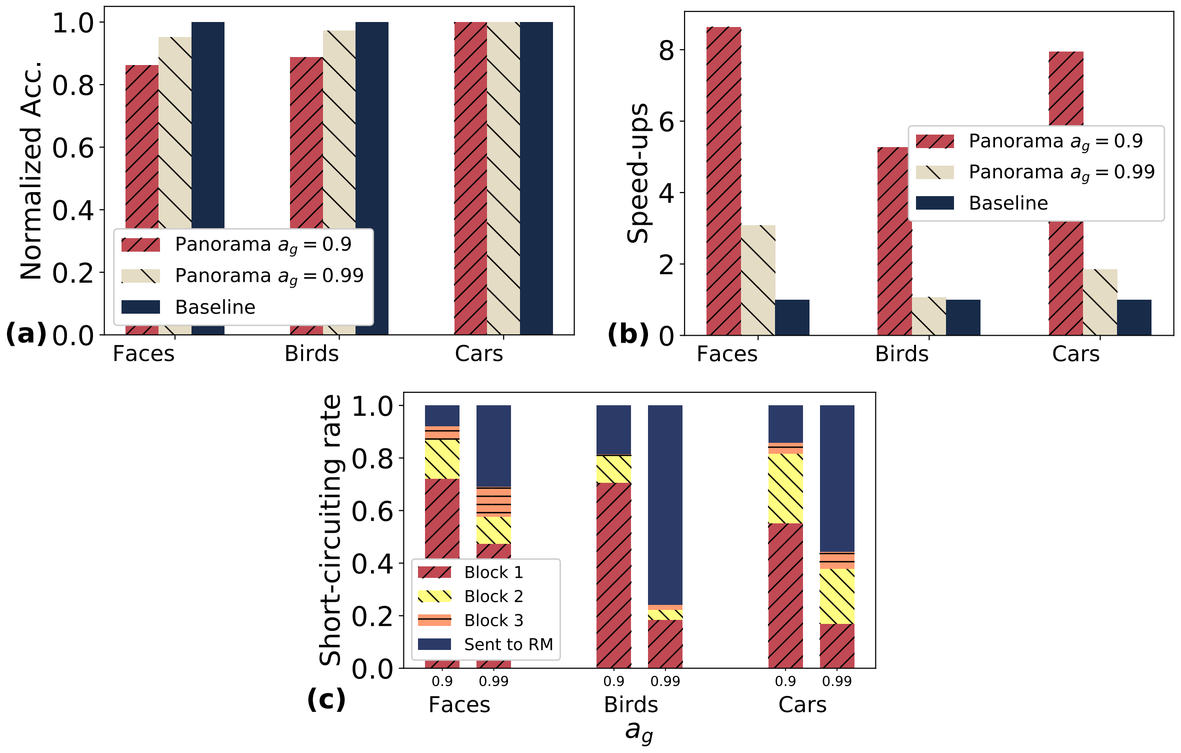


Figure 6.12. End-to-end `in-voc` verification results. (a) Verification accuracy. (b) Relative throughput. Baseline represents the corresponding reference model, the absolute values for three baselines are 7.3, 18.0 and 2.7, respectively; Panorama's results are normalized with respect to them. (c) Fractions of queries short-circuited at each block. "Sent to RM" means those queries were handled by the reference model.

$$K_r = \frac{1}{|\mathbb{Q}|}\sum k_r, \tag{6.5}$$

Software and Hardware. Panorama is implemented entirely in Python. All CNNs and the nearest neighbor search are implemented using TensorFlow 1.4 and Keras 2.1.4 and use GPU acceleration with CUDA 7.0. We used OpenCV 2.0 with FFmpeg backend and PIL 1.1.7 for image preprocessing. All experiments were run on a machine with an NVIDIA GeForce GTX 1080Ti GPU, 8 Intel Xeon E5-2630 v4 cores, and 32 GB RAM.

6.5.2 End-to-end Accuracy and Throughput

We start with the end-to-end performance results, both accuracy, and throughput. Since Panorama is a first-of-its-kind system, prior video querying systems are not quantitatively comparable (more details in Section 6). Thus, we compare Panorama against the reference model, which works only for in-vocabulary queries. On `in-voc test`, we report Panorama's test results relative to the reference model. We then report Panorama's absolute test results on `out-voc test` to show how well it generalizes beyond its supervision vocabulary. We disable the query cache in Panorama for all the experiments in this subsection to let us focus on its main accuracy-throughput tradeoffs. Last we include a strongly supervised video clips dataset *Youtube* on faces to measure Panorama's capability of generalization. The dataset has a vocabulary size of 1595 and over 620k frames in total. We use this dataset to see if Panorama can even generalize beyond its supervision to distinct videos. We test PanoramaNet trained with *Faces* on this dataset. We do not split *Youtube* as it is only used for tests. We poll `known` and simply treat the rest as `out-voc test`.

Verification Queries

Query Set. We randomly sample pairs of frames from `in-voc test` (resp. `out-voc test`) for the in-vocabulary (resp. out-of-vocabulary) verification tests. For all tests, we produce 10^4 pairs each with a 50:50 split for yes and no. Since `out-voc test` in *Birds* is relatively small, we produce only 500 pairs on this but still with a 50:50 yes-no split. We compare two settings for Panorama's accuracy goal configuration parameter: $a_g = 0.9$ and $a_g = 0.99$. All slack parameters (δ_i) are set to 0. Recall that a reference model answers `in-voc` verification via recognition of the objects in both images and comparing their labels, but it does not support `out-voc` verification.

In-Vocabulary Results. Figure 6.12 shows the accuracy and throughput results. We see that Panorama achieves substantially higher throughput while yielding competitive accuracy on *Faces* and *Cars*. For instance, on *Faces*, Panorama with $a_g = 0.99$ has 96% accuracy but is

Table 6.3. `out-voc` verification results. *P: Panorama. †RG: random guessing.

	Faces	*Youtube*	*Birds*	*Cars*
Thrpt. (frames/s)	130	120	96	101
P* accuracy	**81.6%**	**79.6%**	50.0%	**69.7%**
RG† accuracy	50.0%	50.0%	50.0%	50.0%

2x faster; with a 14% drop in accuracy, the other setting is 8x faster. Interestingly, on *Cars*, we found that Panorama's accuracy was slightly higher than the reference model (skipped in the figure, which is capped at 1); recall that we had trained both approaches from scratch on the original labeled dataset in this case. Panorama is also up to 8x faster on *Cars*. On *Birds*, $a_g = 0.9$ is 5x faster while giving 92% of accuracy.

Figure 6.12(c) explains the above results. Panorama's short-circuit processing worked well, with many queries stopping at earlier blocks. In fact, with $a_g = 0.99$, on *Faces*, over half of queries were short-circuited at block 1 and 2. But on *Birds*, more queries were sent to the reference model, yielding a lower average speedup. *Cars* is in between these two extremes. These results validate two of our key design decisions: make PanoramaNet a multi-block architecture that can short-circuit itself and automatically customize it for a dataset to pick an appropriate point in the accuracy-throughput tradeoff.

Out-of-Vocabulary Results. In reality the reference models do not work on out-voc queries. So, we compare Panorama against a random guessing baseline, which is 50% for this binary task. We use $a_g = 0.9$ and no δ_i for all tests. Table 6.3 presents the absolute results. Panorama successfully generalizes beyond its supervision vocabulary to support out-of-vocabulary verification. On *Faces*, the lift is a substantial 33%. *Birds* turns out to be more challenging, while *Cars* falls in between. Panorama's throughput is also well above real-time in all cases. It generalizes well to *Youtube*, which contains distinct videos (e.g. different resolutions, illumination, angles and distances to camera) from *Faces*.

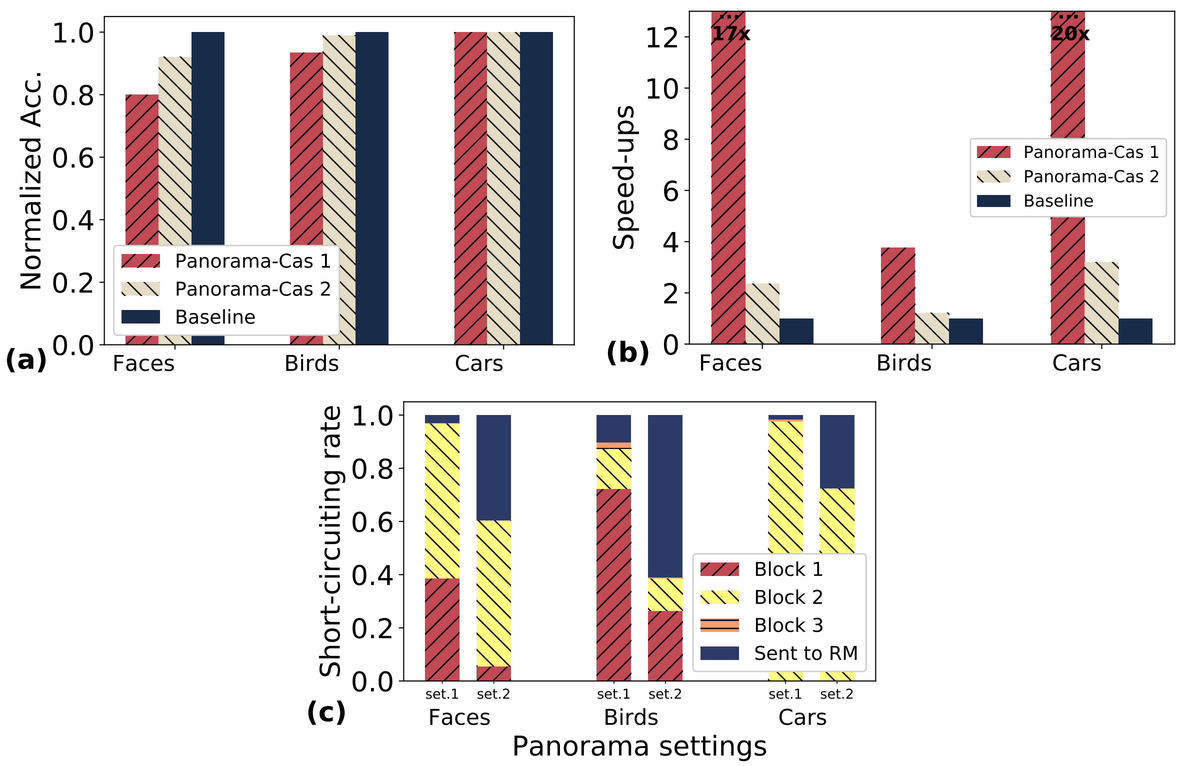


Figure 6.13. End-to-end `in-voc` recognition results. (a) Recognition accuracy. (b) Relative throughput. Baseline represents the corresponding reference model, the absolute values are identical to Figure 6.12; Panorama's results are normalized with respect to them. (c) Fractions of queries short-circuited at each block. "Sent to RM" means being handled by the reference model.

Recognition Queries

Query Set. We compare two settings for Panorama: Cas 1 and Cas 2; Cas 2 represents a stricter accuracy goal than Cas 1, but we vary the configuration parameters across each dataset because they exhibited different properties on the verification tests. For `Faces`, Cas 1 uses $(a_g, \delta_i) = (0.95, 0)$; Cas 2 uses $(a_g, \delta_i) = (0.99, 0.1)$. For `Birds`, Cas 1 uses $(a_g, \delta_i) = (0.9, 0)$; Cas 2 uses $(a_g, \delta_i) = (0.99, 0)$. Finally, for `Cars`, Cas 1 uses $(a_g, \delta_1, \delta_2, \delta_3) = (0.9, \infty, 0, 0)$; Cas 2 uses $(a_g, \delta_1, \delta_2, \delta_3) = (0.99, \infty, 0, 0)$. Note that setting $\delta_j = \infty$ means short-circuiting is not allowed at Block$_j$.

In-Vocabulary Results. Figure 6.13 shows the results. On *Faces*, Cas 1 is 17x faster, while offering almost 80% relative accuracy, Cas 2 is 2x faster while yielding 92% accuracy. On

Table 6.4. `out-voc` recognition results. *P: Panorama. †RG: random guessing. ‡: Top-1 accuracy.

	Faces	*Youtube*	*Birds*	*Cars*
Thrpt. (frames/s)	107	105	97	63
P* Accuracy	**74.5%**	**46.4%**	**73.9%**‡	**49.6%**
RG† accuracy	38.5%	0.3%	50%‡	5.7%

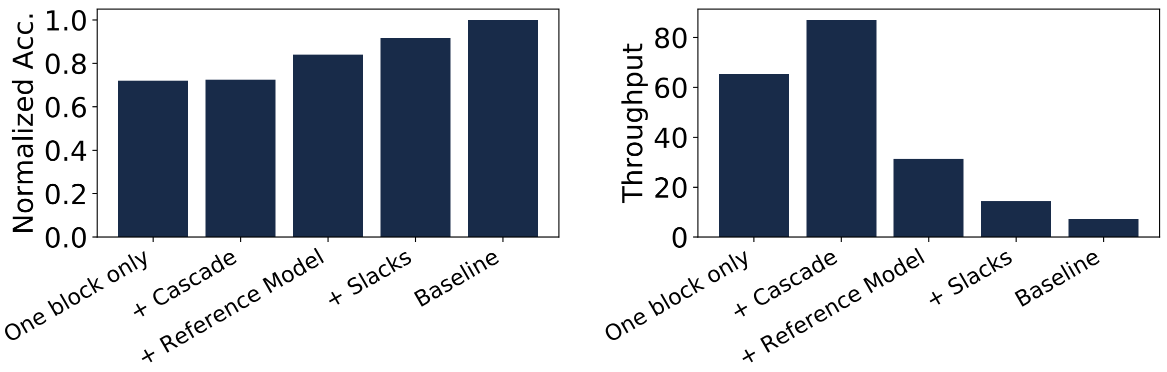


Figure 6.14. Factor analysis. Accuracy is normalized against the reference model.

Cars, both settings match (or slightly surpass) the reference model's accuracy, while being up to 20x faster. Compared to *Faces*, *Birds* offers a slightly more modest speedups but with higher accuracy. Figure 6.13(c) explains these results in terms of the short-circuiting results. We see similar behaviors as in the `in-voc` verification tests.

Out-of-Vocabulary Results. Once again, since the reference models are not applicable here, we compare Panorama to a random guessing baseline. We use $a_g = 0.9$ and no slacks for all tests. Recognition is effectively multi-class, not binary. So, the accuracy of random guessing depends on the sizes of the vocabularies in `known`; these sizes are 14, 2, 87, 1595 for *Faces*, *Birds*, *Cars*, *Youtube*, respectively. We report top-1 accuracy for *Birds* (since the vocabulary size is only 2) and top-5 accuracy for the rest. Table 6.4 presents the absolute results. Once again, we see that Panorama successfully generalizes beyond its supervision vocabulary to support out-of-vocabulary recognition queries too. On *Cars*, the lift is a substantial 44%. It generalizes well to *Youtube*, offering 46% lift on accuracy with high throughput.

Table 6.5. Impact of λ_k on block-wise verification acc.

λ_k	Acc. Block1	Acc. Block2	Acc. Block3
1:1:1	79.9%	87.8%	89.1%
8:2:1	83.2%	84.3	87.2%
100:10:1	61.4%	58.4%	65.5%

6.5.3 Drill-down Analysis

Factor Analysis. We now drill into Panorama's behavior to show the effects of its various components on the throughput-accuracy tradeoff. We expand `in-voc` recognition tests on *Faces* for this purpose. We use Cas 2 described above for the cascade related configs. Figure 6.14 presents the results of each component's effect. We start by disabling short-circuiting and taking only the output of last block of PanoramaNet. This provides over 60 FPS but limits the accuracy to 72%. If we enable the rest blocks and cascaded processing, throughput boosts to over 80 FPS. This demonstrates the effeteness of our cascade. Next if we concatenate the reference model into the cascade, the accuracy further improves to 84% with the speedups drop to 30 FPS. The last element needed are the slacks to yield 92% accuracy, with a 2x speed-up compared to baseline still.

Impact of λ_k. We now investigate the impact brought by different settings of λ_k. We vary these weights and train three different models and report the raw verification performance of each block of the models on our *Faces* `val` split. Table 6.5 summarizes the results. Compared to no weights (1:1:1), setting a higher weights on the first block (8:2:1) does improve the individual performance of block1, however, the subsequent blocks loses some accuracy. These weights provide a trade-off between early and later block discrimination power. On the other hand, too large weights (100:10:1) interferes with the training process and fails to converge.

6.5.4 Query Cache and Scalability Test

We now stress test Panorama's throughput by raising the size of the known objects for recognition queries. Some real-world video monitoring applications could indeed have to deal

Table 6.6. Impact of query cache on recognition acc.

Cache size	Relative accuracy	Cache hit rate
0(No cache)	80%	0%
1	80%	43%
10	80%	80%
100	80%	89%

Table 6.7. Results of the scalability test. W/O means Panorama without cache, Cache X means Panorama with cache size X. All values in the right four columns are throughputs reported in frames/sec.

|`known`|	Baseline	W/O cache	Cache 1	Cache 100
10^5	7.2	24.0	33.0	**45.9**
5×10^5	6.1	9.0	11.4	**20.7**
10^6	4.6	5.1	6.7	**14.7**

with millions of objects, e.g., identifying faces in mall security surveillance, and the database for faces can be excessively large. We pick the same setting on the in-voc recognition test of *Faces* and use cascade setting 1.

Impact of query cache on accuracy. We now enable the cache and investigate the impact brought by the query cache with varying cache size. Table 6.6 summarizes the results. As the size of the cache goes up, the cache hit rate rises, while the accuracy remains relatively constant, meaning this cache does not influence accuracy much. Although the video is after downsampling, the cache hit rate can still be as high as 89%. This demonstrates the temporal redundancy characteristics of video.

Scalability test. To simulate the case where the `known` set is at scale, we enlarge the existing `known` set with duplicates. For this experiment, we run Panorama in three modes: without the query cache, with size-1 cache and size-100 cache. We compare the results to the *Faces* reference model, which also yields embeddings and uses k-nn for recognition. Table 6.7 shows the results for throughput when `known` is at scale. In this scenario the k-nn search becomes a bigger bottleneck compared to CNN inference. Without the cache, the throughput of Panorama will eventually join baseline as the known object set expands. However, Panorama with size-100 cache still offers 3x$\sim$6x speedups depending on $|\texttt{known}|$. Cache-100 also outperforms Cache-1, as the former has much higher cache hit rate and skips more searches. This validates the benefits of the query cache for large-scale recognition queries.

6.5.5 Conclusion

The success of deep CNNs presents new opportunities for querying video data. However, most off-the-shelf models and video querying systems assume the prediction vocabulary is bounded, impeding many emerging video monitoring applications. In response, we present a new data system architecture, Panorama, for unbounded vocabulary querying of video. Panorama saves users the hassle of retraining models post deployment as the vocabulary grows. It is based on a multi-task and unified architecture, and its deployment is end-to-end automated and domain-agnostic. Relative to bespoke domain-specific models, Panorama's unified system offers competitive accuracy but with higher throughput.

Panorama generalizes beyond a given finite vocabulary to unseen objects of the same type in a given domain. This is a form of subclassing, i.e., Panorama does not generalize to new types of objects or new domains. We now offer some insights on when Panorama may or may not be applicable. It applies to fine-grained visual tasks, and the granularity is determined by the supervision provided. The viability of a task depends on the availability of large-scale datasets and/or high-quality reference models and the degree of difficulty of the task itself. Faces are most viable because of their relatively well-understood properties: mostly 2-D and simple geometric layout. There are also many large datasets for faces. For cars, the datasets are decent; so, the accuracy is good. As for other domains, as long as there exists large-enough fine-grained datasets and/or good reference models, we believe Panorama is applicable.

Chapter 6 contains material from "Panorama: A Data System for Unbounded Vocabulary Querying over Video" by Yuhao Zhang and Arun Kumar, which appears in Proceedings of

VLDB Endowment Volume 13, Issue 4, Decemember 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Related Work

7.1 Related Work for CEREBRO

Systems for Model Selection. Google Vizier [104], Ray Tune [183], Dask Hyperband [260], SparkDL [75], and Spark-Hyperopt [129] are systems for model selection. Vizier, Ray, and Dask-Hyperband are pure task-parallel systems that implement some AutoML procedures. SparkDL and Spark-Hyperopt use Spark for execution but distribute configs in a task-parallel manner with full data replication PANORAMA offers higher overall resource efficiency compared to pure task- or pure data-parallel approaches.

AutoML Procedures. AutoML procedures such as Hyperband [176] and PBT [130] are orthogonal to our work and exist at a higher abstraction level. They fit a common template of per-epoch scheduling in PANORAMA. While ASHA [175] does not fit this template, PANORAMA can still emulate it well and offer similar accuracy. Bayesian optimization is a class of AutoML procedures, some of which have a high degree of parallelism for searching configs (e.g., Hyperopt [41]); PANORAMA supports such procedures. Some others run a sequential search, leading to a low degree of parallelism (e.g., [151, 40]); these may not be a fit for PANORAMA.

Distributed SGD Systems. There is much prior work on data-parallel distributed SGD, including centralized fine-grained (e.g., [287, 328, 137, 127]) and decentralized fine-grained (e.g., [291, 182, 287]). These are all complementary to our work because they train one model at a time, while we focus on parallel model selection. As we showed, such approaches have higher communication complexity and thus, higher runtimes than MOP in our setting. Also, since PANORAMA performs logically sequential SGD, it ensures theoretically best convergence efficiency. CROSSBOW [153] proposes a new variant of model averaging for single-server multi-GPU setting. But it is also complementary to our work, since it also trains one model at a time. Overall, our work breaks the dichotomy between data- and task-parallel approaches, thus offering better overall resource efficiency.

Hybrid Parallelism in ML Systems. MOP is inspired by the classical idea of process migration in OS multiprocessing [32]. We bring that notion to the data-partitioned cluster setting. This generic idea has been used before in limited contexts in ML systems [158, 46]. The closest to our work is [65], which proposes a scheme for training many homogeneous CNNs on a homogeneous GPU cluster. They propose a ring topology to migrate models, resembling a restricted form of MOP. But their strong homogeneity assumptions can cause stalls in general model selection workloads, e.g., due to heterogeneous neural architectures and/or machines. In contrast, we approach this problem from first principles and formalize it as an instance of open shop scheduling. This powerful abstraction lets PANORAMA support arbitrary deep nets and data types, as well as heterogeneous neural architectures and machines. It also enables PANORAMA to support replication, fault tolerance, elasticity, and arbitrary AutoML procedures, unlike prior work. SystemML also supports a hybrid of task- and data-parallelism for better plan generation for linear algebra-based classical ML on top of MapReduce [48]. PANORAMA is complementary due to its focus on deep nets and SGD's data access pattern, not linear algebra-based classical ML. Finally, a recent benchmark study suggested that communication bottlenecks inherent in pure data-parallelism imply hybrid parallelism is crucial for scalable ML systems [269]. Our work validates that suggestion for deep learning workloads.

Multi-Query and Other System Optimizations. MOP is also inspired by multi-query optimization (MQO) [256]. A recent line of work in the database literature studies MQO for deep learning, including staging and sharing work in CNN transfer learning [208] and batched incremental view maintenance for CNN inference [210, 225, 211]. PANORAMA furthers this research direction. All these MQO techniques are complementary and can be used together. Several works optimize the internals of deep net or SGD systems, including communication-computation pipelining [216], new compilation techniques [136], model batching [218], and execution on compressed data [171]. They are complementary to PANORAMA, since they optimize lower-level issues. MOP's generality enables PANORAMA to be hybridized with such ideas.

Scheduling. Gandiva [298], Tiresias [110], and SLAQ [312] are cluster scheduling frameworks for deep learning. They focus on lower-level primitives such as resource allocation and intra-server locality for reducing mean job completion times. PANORAMA is complementary as it exists at a higher abstraction level and focuses on model selection throughput. How compute hardware is allocated is outside our scope. There is a long line of work on job scheduling in the operations research and systems literatures [121, 55, 100]. Our goal is *not* to create new scheduling algorithms but to apply known techniques to a new ML systems setting.

7.2 Related Work for CEREBRO on Data Systems

ML in Data Systems. There is a long line of work on ML in data systems. The general approach is to implement ML algorithms via UDFs or other APIs exposed by the data system. Apache MADlib [120, 91] is one of the most mature such tools. The UDAF approach we studied for integrating MOP is already a part of MADlib. Vertica-ML [87], Oracle Machine Learning [16], Microsoft SQL Server ML Services [14], and Google BigQuery [9] are other prominent examples of in-RDBMS ML tools. [235] brings ML to column stores. MLlib [200] and MLlib* [321] use Spark's APIs to implement various ML algorithms. Mahout [30] is a distributed ML system on top of dataflow systems. Increasingly, more data system builders want to integrate with DL via wrappers that invoke popular DL tools: Horovod on Spark [11], TensorFrames [18], and PS2 [320] are examples. More generally, the DBMS and cloud industry believe DBMSs will continue to play a key role in enterprise ML [23].

Some works also expand DBMS support for ML. Raven [144] deeply integrates ML runtimes into a DBMS. UDA-GIST [174] expands support for algorithms that are both data-parallel and state-parallel. [194] adds linear algebra support to RDBMS. [308] proposes a "tensor-relational" algebra towards declarative ML. TensorDB [147] is a system for in-DBMS tensor decomposition. [146] focuses on in-DBMS sparse tensors for ML. DB4ML [133] expedites iterative ML algorithms via asynchrony. [99] discusses declarative model weights distribution/aggregation for data-parallel ML. [132] adds better support for recursion to RDBMS for distributed ML. MLearn [255] is a declarative language for in-DBMS ML. AIDA [83] provides an abstraction for in-DBMS data analytics; it uses DBMS for relational operations and embeds Python for linear algebra.

All of the above works are complementary to ours. To the best of our knowledge, our paper is the first to study system design alternatives and tradeoffs for enabling DL workloads on DBMSs. Specifically, we focus on bringing a recently published hybrid parallel execution approach for DL model selection, MOP, to the traditionally bulk-synchronous parallel world of DBMSs.

Custom ML Systems. There is also a long line of work on custom systems for ML training/model selection. FlexPS [127] and Lapse [245] are both optimizations to Parameter Server [177]. Horovod [257] brings in decentralized communication to boost runtime efficiency. Vizier [104] and Rafiki [288] are systems for task-parallel model selection; Ray [206, 184] was initially designed for reinforcement learning but recently also supports task-parallel model selection. Singa [287, 223] and SystemML [45, 48, 47] are end-to-end platforms for ML that supports various distributed training. Visus [250] and Ease.ml [244, 243] are examples of AutoML systems that manage the whole ML lifecycle, including both data management and model selection. Crossbow [153] and Ako [291] are systems for better resource scheduling and utilization for ML. [79] handles collaborative working environments for ML development. Litz [234] focuses on the elasticity of distributed ML.

All these works are also complementary to ours because they study standalone ML/DL execution, not integration with data systems. While some of them may be faster than in-DBMS ML tools, as we explained in depth in this paper, ML practitioners, especially in enterprises, grapple with a more complex Pareto frontier beyond just runtimes. Our paper lays out these tradeoffs in bringing DL workloads closer to DB-resident data. That said, the CTQ approach we studied was in part inspired by the pervasive use of task parallelism in such custom ML systems, including in Cerebro as we explained earlier. More generally, we believe these historically distinct work lines–custom ML systems and ML on data systems–can learn a lot from each other.

Data Access and Pipeline Optimizations for ML. There is much prior work on optimizing ML+data processing pipelines. Lara [162], Alpine-Meadow [259], and KeystoneML [262] all allow the user to define pipelines with their APIs and perform pipeline-level optimizations. Helix [300] injects intelligent caching and reuse between training iterations to reduce redundant work. [84] proposes linear algebra that could work upon compressed data, thus saving decompression time. [171] introduces a tuple-oriented compression scheme for matrix and mini-batch SGD computations directly on compressed data.

The above works are largely orthogonal to our paper, since our goal is *not* to devise novel optimization schemes or systems but rather to analytically and empirically study the tradeoffs of alternative approaches to bring DL workloads to DB-resident data. That said, the DA approach we studied was in part inspired by such prior work on ML operating more directly on the raw stored data. It is interesting future work to integrate more such optimizations into systems that bring DL closer to DB-resident data.

Data Management for ML. More generally, data management for ML is a hot and pressing research topic [42, 253, 159, 233]. Such works aim to optimize or automate data management tasks in ML workflows to reduce user burden. Data Programming [240] and Snorkel [238] focus on ML training data creation through weak supervision and generative models. DeepDive [310] is a system for knowledge base construction. ModelDB [275, 274], TFX [37], Mlog [180], and MLFlow [204] all add data management and model management support for ML. ARDA [71] uses DBMS for data augmentation and feature selection tasks. Activeclean [155], boostclean [154] and [190] focus on data cleaning and debugging for ML. Vamsa [215] supports data lineage tracking for Python ML scripts. [115] proposes the concept of model materialization and reuse to speed up ML training.

All these works are also largely orthogonal to ours, since our focus is specifically on tradeoffs of in-DBMS execution of DL model selection, not auxiliary data/model management capabilities. Lessons from our work can be easily integrated with these other tools to enhance end-to-end support for ML applications for DB users.

7.3 Related Work for LOTAN

GNN Systems. Many systems have been proposed to tackle the efficiency and scalability challenges of GNN training. Our work differs from them in our fundamental architecture design of separation of graph and neural network and our technical innovations. We have also conceptually compared them in Section 5.2.3 and tested against some of the most related and state-of-art systems in Section 5.7. Most of their techniques are complementary to our work. DGL [285, 323], and PyG [93] are prominent examples of all-purpose GNN frameworks designed for generality and usability. AliGraph [326], GraphScope [301], and PSGraph [138] are GNN systems designed for industry-scale usage with an emphasis on sampling-based GNN training, which differs from Lotan's focus on full-batch training. NeuGraph [195] is one of the first systems to incorporate GNNs into an extended Gather-Apply-Scatter framework. It provides a scheduling scheme for shipping models/data in and out of multiple GPUs; its techniques are largely complementary to our work.

Other GNN systems proposed techniques ranging from memory management, communication reduction, approximated processing, and disk spilling. PaGraph [186] utilizes spare GPU memory for data caching to boost speed when the workload is relatively small. P3 [98] separates the graph metadata and graph properties and places them in a way to reduce communications.

Sancus [230] proposes a communication reduction scheme via historical gradient caching and update skipping. Similarly, PipeGCN [281] uses pipeline parallelism with stale updates to speed up GNN training. They largely focus on the efficiency of GNN training via approximated processing and assumes the model and data can comfortably fit in GPU memory. PaGraph, P3, and Sancus are largely orthogonal to our work as Lotan is designed for large workloads, we do not assume an abundance of GPU memory. Dorylus [270] employs serverless functions to explore monetary cost-efficiency; our system is still for provisioned clusters, and we rely on existing data systems instead of custom-built ones used in Dorylus. Roc [135] uses main memory as swapping space to offload over-the-size data from GPU. MariusGNN [280] further proposes disk-spilling to increase the effective memory size. These techniques complement Lotan, and by employing a secondary-storage-aware graph data system, Lotan can naturally piggyback on its disk spilling capability. ALG [315] is designed for active learning setup which is largely orthogonal to our work. G3 [187] proposes to substitute DL frameworks with GPU-based graph operations; it can potentially be a candidate for the DL Engine in Lotan and is complementary to our work.

Graph Analytics Systems. Prior to the GNN era, large-scale systems were built for non-GNN graph analytical workloads and data management. Ranging from graph DBMS [80, 219, 92, 164], to classical graph analytics systems [198, 106, 105, 103, 152, 31, 267], and Graph Embedding learning (not to be confused with GNN) systems [170, 205, 296]. They are generally orthogonal to our work, as they target very different sets of workloads, and they seldom work with GNNs. Most techniques are workload-specific and not directly applicable to GNN training, but some may be complementary.

Faster and More Scalable GNNs. Ever since the first wave of GNN models arrived, algorithmic research has been active in tackling some of the scalability issues of GNNs by approximated processing and simplified architectures. This line of research is orthogonal to our work, as our goal is not to propose any new GNN model architecture but instead focus on the fundamental system challenges that will not be fixed by GNN model research alone.

GraphSage [113] proposes mini-batch training and neighborhood sampling to reduce the data dependencies. FastGCN [67] runs even more aggressively IID sampling on the graph by directly controlling the number of nodes involved. SGC [294] challenges GNNs by proposing trivial two-layer architectures that reportedly could offer a similar accuracy performance. EIGNN [188] further extends SGC to an infinite depth model and uses eigendecomposition to boost efficiency. Graph coarsening techniques [128] have also been explored to preprocess and down-sample the input data.

7.4 Related Work for PANORAMA

Vision and Label-efficient ML. We already explained how Panorama relates to prior works in vision, including task definitions (Section 2.1), CNN-based vision (Section 2.2), how PanoramaNet is based on lessons of recent CNNs (Section 4.1), deep supervision (Section 4.2), and which CNNs act as reference models (Section 5). Thus, we now only discuss a key aspect of Panorama's goal that is related to several lines of work in ML. Deep CNNs typically need large labeled datasets, but many applications may not have so much labeled data. To meet this challenge, the ML community has long worked on label-efficient learning schemes, including *zero-shot* [167, 166, 25, 96, 220], *one-shot* [165, 88, 89, 35, 95, 167], and *open-set* [251, 252, 97, 38] learning. Zero-shot learning asks models to recognize new unseen classes by transferring semantic knowledge learned from training classes, often via auxiliary metadata for retraining. One-shot learning relaxes this assumption by asking for one or a few labeled examples per new class. Open-set learning also aims to remove the closed-world vocabulary assumption, but it does so by retraining models to recognize both old and new classes. Such alternative learning schemes are sometimes collectively called *life-long* learning [69, 231, 271].

All these previous efforts in ML inspire our formulation of the *unbounded vocabulary* problem, but our goal is *not* proposing new learning schemes, vision tasks, or more accurate

CNNs. Our focus in Panorama has a crucial system functionality difference aimed at benefiting users of video monitoring applications.

Cascaded Classification. Cascaded models have long been used in multimedia systems to improve accuracy and efficiency. Introduced in the Viola-Jones object detection framework [277], recent works have extended this idea to deep CNNs [58, 114, 173, 265, 282, 59, 125]. These works inspired our design decision of making PanoramaNet cascaded, but our approach extends this idea along two lines: we fuse it with multi-task learning for unified processing instead of disparate bespoke models and we use deeply supervised training (originally designed to improve accuracy [169]) to make this fusion possible. Our short-circuiting configuration also supports a more tunable accuracy-throughput tradeoff.

Multimedia Databases. The multimedia database community has long studied content-based image retrieval (CBIR), whose goal is to retrieve images or videos from a database that have the same "content" as a query image [22, 141, 303, 316, 139, 232, 134, 261]. The notion of content is application-specific. Early CBIR works used hand-crafted vision features (e.g., SIFT) but recent ones showed that CNN features improve accuracy. Panorama's focus is *not* on CBIR but rather video monitoring applications. That said, our design decision of using embedding extraction to tackle unbounded vocabularies is inspired by work on CBIR. To the best of our knowledge, ours is the first work to exploit this connection between multimedia DB techniques and video monitoring.

Video Querying Systems. Video monitoring systems have seen a resurgence of interest in the DB and systems literature. NoScope [142] creates a model cascade with simple filters and a specialized cheaper CNN to improve querying efficiency compared to a larger reference CNN. Focus [123] splits video monitoring into ingesting and query stages to enable more accuracy-efficiency tradeoffs, including indexing objects offline and using them to speed up queries. BlazeIt [143] proposes an SQL-like language for selection and aggregation queries over frames and uses approximation techniques to improve efficiency. All these systems support only binary or finite multi-class vocabularies, which make them complementary to Panorama. Nevertheless, our work on Panorama was inspired by these systems, and we fundamentally expand video monitoring functionality to unbounded vocabularies while ensuring system efficiency.

Among video analytics systems, CaTDet [199] reduces inference costs by computing regions of interests based on historic detections. FilterForward [61] uses constrained edge nodes better. VideoStorm [311] and Optasia [193] are large-scale video analytics systems that aim to reduce latency. RAM^{3}3S[36], KDEDisStrOut[324], and other research[266] aim at real-time video analytics from massive multimedia streams. All these systems are orthogonal to Panorama, since they focus on better resource management and parallelism for analytics queries, not enabling unbounded vocabularies for monitoring queries. We believe Panorama can be integrated with such systems in the future.

Chapter 8

Conclusion and Future Work

In this dissertation, we reimagine DL systems as DL data systems; we innovate upon many database-inspired techniques, such as multi-query optimizations, query plan rewrites, approximate processing, and multimedia databases, and apply them to improve DL systems for three representative workloads: model selection, training, and inference on various data modalities. Our work offers runtime efficiency gains and improvements in system scalability over state-of-the-art solutions. We also make the first comprehensive attempts to bring DL to database-resident data. We have demonstrated that bridging the gap between existing data systems and DL workloads, without modifying any existing codebase and infrastructure or introducing overheads, is possible. Our work is an important step towards the goal of practical, scalable, and high-throughput DL data systems. It opens up design freedom, saves DL practitioners' costs, and provides viable guides and insights for data systems researchers.

8.1 Future Work Related to CEREBRO and CEREBRO on Data Systems

Model Parallelism. CEREBRO is a hybridization of task parallelism and data parallelism. In addition, model parallelism exists to partition and execute a large model across multiple GPUs or computational nodes, primarily because the model does not fit in a single processing unit. Today's large models rely on this technique to train on dozens to hundreds of GPUs. However, these models are not free from the model selection problems, and the costs associated become astronomical as these models are extremely demanding in computational power. It remains an open question on how to build a scalable distributed system that can optimize the workloads holistically with both model hopper parallelism and model parallelism.

Data Transfer Between ML and Data Systems. We realized that the current data warehouse architecture lacks optimizations for distributed DL. One of the major problems is the data format: data warehouses store data in proprietary pagefile formats. Accessing data via the DBMS can bring severe bottlenecks for DL workloads, which require frequent and rapid table scans. It is largely an open research question, but some proposals are promising, such as Lakehouse [309]. In this work, we devised Direct Access with caching, and we hope it can serve as a candidate solution to the above problems. However, DA is coupled with the proprietary DBMS data formats. Standard pagefile formats such as Parquet would simplify the implementation and increase the portability drastically. Similarly, in-memory formats like Apache Arrow would vastly simplify the data transmission process between different runtimes and may also bring performance boosts.

8.2 Future Work Related to LOTAN

Large Models and Model Parallelism. GNNs, are still DL models, albeit with a distinctive data access pattern. With the tremendous success of Large Models (Foundation Models) in computer vision, natural language processing, and chatbots, it is only natural to imagine the same set of techniques can be applied to graph data as well. A huge gap exists between the model size of popular GNNs and the SOTA CNNs or Transformers, and it hints at vast opportunities for performance gains. Efforts already exist to hybridize Transformers with GNNs [322]. However, the research frontier has been primarily limited by the costly and inefficient execution of GNNs, and no truly Large GNN models exist. Therefore, to facilitate Large GNNs research and open up new design freedom for the practitioners, the systems community needs to investigate the abstractions, optimizations, and efficient executions of Large GNNs requiring a novel fusion of both graph processing and model parallelism, which adds an extra complexity on top of the GNNs' data parallelism.

High-throughput GNN Inference. So far, there has been very little work in accelerating GNN inference, albeit inference takes up the majority of costs once the model is trained and deployed. For an in-data-system GNN system such as LOTAN, the inference is equally, if not more important than training. Both opportunities and challenges exist: inference tasks have relaxed requirements on accuracy, enabling approximate processing techniques for acceleration. At inference time, the model and many intermediate states also become static, and caching them would lead to performance boosts unavailable for training. However, unlike training, which typically uses a static graph, the graph may become dynamic at inference time. The system must be able to cope with the added complexity of dynamic graphs and provide intelligent caching and re-computing mechanisms to maximize the runtime performance.

8.3 Future Work Related to PANORAMA

More Types of Queries. Beyond the identity recognition and verification queries that PANORAMA targets, various tasks such as motion recognition and cross-camera object tracking exist. The same unbounded vocabulary problem remains, and the same set of proxy model methods still applies. In future work, the definition of unbounded vocabulary problem needs to be expanded to include these tasks, and new proxy model architectures need to be developed to answer these queries.

Improving Reference Models and PanoramaNet. The landscape of computer vision has been shifting constantly. Recent advances such as Vision Transformers (ViT) [82] have emerged to supersede regular CNNs. These models have a clear advantage over CNNs in terms of accuracy but are more demanding in terms of computational power, making them premium candidates to serve as Reference Models. It remains an open question of how to bring Transformers into PanoramaNet and retain the cascaded nature and the associated deep supervision of the latter. Further, it is left for future work to re-design PanoramaNet so that the sequential nature of the cascade processing can be relaxed and increase the degree of parallelism.

Appendix A

CEREBRO

A.1 CEREBRO API Usage Example

In this Section, we present a detailed example on how the PANORAMA API can be used to perform the *ImageNet* model selection workload explained in Section 3.6.1.

Before invoking the model selection workload users have to first register workers and data. This can be done as per the API methods shown in Listing 1 and Listing 2.

Listing A.1. Registering Workers

```
#### API method to register workers ###
# worker_id:    Id of the worker
# ip       :    worker IP
#
# Example usage:
#   register_worker(0, 10.0.0.1)
#   register_worker(1, 10.0.0.2)
#   ....
#   register_worker(7, 10.0.0.8)
#######################################
register_worker(worker_id, ip)
```

Listing A.2. Registering Data

```
## API method to register a dataset ###
# name           : Name of the dataset
# num_partitions : # of partitions
#
# Example usage:
#       register_dataset(ImageNet, 8)
#######################################
register_dataset(ImageNet, 8)
```

```
## API method to register partition ###
##         availability             ###
# dataset_name  : Name of the dataset
# data_type     : train or eval
# partition_id  : Id of the partition
# worker        : Id of the worker
# file_path     : file_path on the
#                 worker
#
# register_partition(ImageNet, train,
#        0,
#        0, /data/imagenet/train_0)
#######################################
register_partition(dataset_name,
         data_type,
         partition_id, worker,
         file_path)
```

Next, users need to define the initial set of training configurations as shown in Listing 3.

Listing A.3. Initial Training Configurations

```
S = []
for batch_size in [64, 128]:
  for lr in [1e-4, 1e-5]:
    for reg in [1e-4, 1e-5]:
      for model in [ResNet, VGG]:
        config = {
          batch_size: batch_size,
          learn_rate: lr,
          reg: reg,
          model: model
        }
        S.append(config)
```

Users also need to define three functions: *input_fn*, *model_fn*, and *train_fn*. *input_fn* as shown in Listing 4, takes in the file path of a partition, performs pre-processing, and returns in-memory data objects. Inside the *input_fn* users are free to use their preferred libraries and tools provided they are already installed on the worker machines. These in-memory data objects are then cached in the worker's memory for later usage.

Listing A.4. *input_fn*

```
#### User defined input function ######
# file_path :   File path of a local
#               data partition
#
# Example usage:
#    processed_data = input_fn(file_path)
```

```
#######################################
def input_fn(file_path):
        data = read_file(file_path)
        processed_data = preprocess(data)
        return processed_data
```

After the data is read into the worker's memory, PANORAMA then launches the model selection workload. This is done by launching training units on worker machines. For this PANORAMA first invokes the user defined *model_fn*. As shown in Listing 5, it takes in the training configuration as input and initializes the model architecture and training optimizer based on the configuration parameters. Users are free to use their preferred tool for defining the model architecture and the optimizer. After invoking the *model_fn*, PANORAMA injects a checkpoint restore operation to restore the model and optimizer state from the previous checkpointed state.

Listing A.5. *input_fn*

```
#### User defined model function ######
# config : Training config.
#
# Example usage:
#       model, opt = model_fn(config)
#######################################
def model_fn(config):
    if config[model] == VGG:
        model = VGG()
    else:
        model = ResNet()

    opt = Adam(lr=config[learn_rate])
    return model, opt
```

After restoring the state of the model and the optimizer, CEREBRO then invokes the user provided *train_fn* to perform one sub-epoch of training. As shown in Listing 5, it takes in the data, model, optimizer, and training configuration as input and returns convergence metrics. Training abstractions used by different deep learning tools are different and this function abstracts it out from the CEREBRO system. After the *train_fn* is complete the state of the model and the optimizer is checkpointed again.

Listing A.6. *input_fn*

```
#### User defined train function ######
```

```
# data      : Preprocessed data
# model     : Deep learning model
# optimizer : Training optimizer
# config    : Train config.
#
# Example usage:
#       loss = train_fn(data, model,
#                   optimizer, config)
######################################
def train_fn(data, model, optimizer, config):

    X, Y = create_batches(data,
                config[batch_size])

    losses = []
    for batch_x, batch_y in (X,Y):
        loss = train(model, opt,
            [batch_x, batch_y])
        losses.append(loss)

    return reduce_sum(losses)
```

For evaluating the models, we assume the evaluation dataset is also partitioned and perform the same process. We mark the model parameters as non-trainable before passing it to the *train_fn*. After a single epoch of training and evaluation is done, CEREBRO aggregates the convergence metrics from all training units from the same configuration to derive the epoch-level convergence metrics. Convergence metrics are stored in a configuration state object which keeps track of the training process of each training configuration. At the end of an epoch, configuration state objects are passed to the *automl_mthd* implementation for evaluation. It returns a set of configurations that needs to be stopped and/or the set of new configurations to start. For example in the case of performing Grid Search for 10 epochs, the *automl_mthd* will simply check whether an initial configuration has been trained for 10 epochs, and if so it will mark it for stopping.

A.2 CNN Compute Costs

Table A.1 lists the computational costs of the CNNs used for the simulation experiment which compares different scheduling methods. These costs were obtained from a publicly available benchmark[1].

[1]https://github.com/albanie/convnet-burden

A.3 Straggler Issue in Celery

One potential issue that could impact task-parallel systems' performance is load balancing. Given the large variance of runtimes for deep-nets training, the scheduling generated by Celery could lead to severe straggler issues that impairs the end-to-end runtime of the whole workload. On the other hand, PANORAMA suffers far less from this problem because it operates on a finer granularity; our tasks are chunked into sup-epochs and hence it is less likely for long-running stragglers to appear.

We take the Criteo tests showed in Section 3.6.1 as example. Without any prior or domain knowledge, it is impossible to know the runtime of each task before-hand and therefore Celery could schedule a plan like Figure A.1. The execution suffers from the straggler config#0 and needs 27.4 hrs to run.

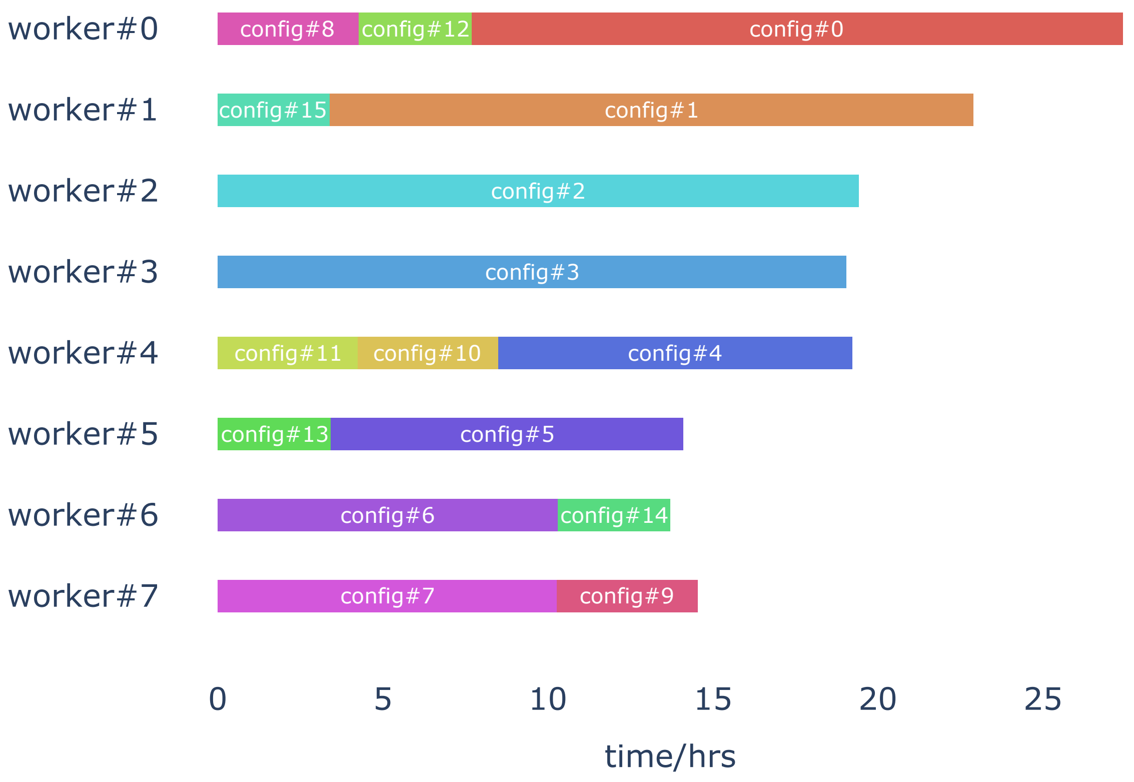


Figure A.1. An unbalanced work schedule generated by Celery for Criteo tests.

However, if with a proper estimation/profiling of the runtimes/workloads, it is possible to fix this straggler issue with a carefully curated schedule as showed in Figure A.2. This schedule drastically reduces the runtime to 19.7 hrs.

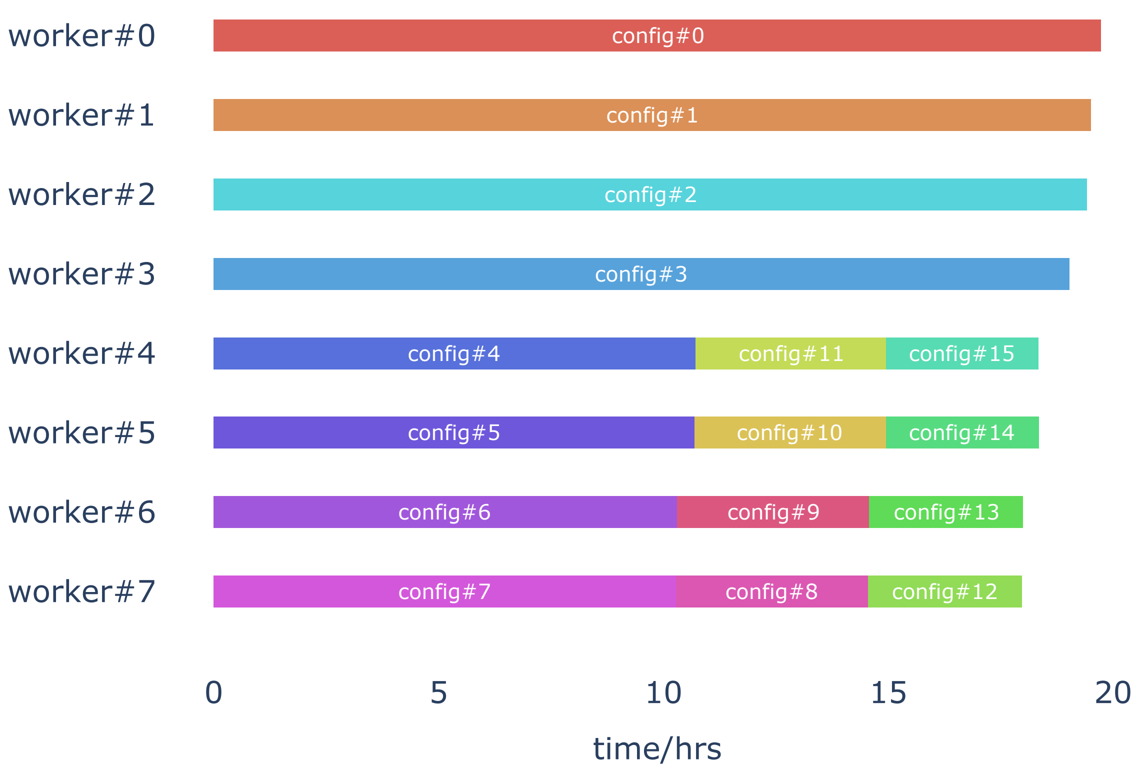


Figure A.2. Best possible work schedule with Celery for Criteo tests.

In Section 3.6.1, we decided to show the runtime with the best-possible scheduling for Celery, as we do not wish to unfairly punish the adversarial systems, and load balancing/runtime estimation of deep learning workloads are out of the scope of this paper. We believe these decisions can ultimately help the reader focus on the benefits and advantages of our system.

A.4 Extension: Horovod Hybrid

A typical model selection workflow begins with a large number of model configs, and narrows down the scope gradually over epochs, ending up with a handful of model configs to train till convergence. It means that at the later stages, we may encounter scenarios where the number of model configs, $|S|$, can be smaller than the number of workers, p. In these scenarios PANORAMA can lead to under-utilization of the cluster.

We mitigate this issue by doubly hybridizing MOP with data parallelism. To this end, we implement a hybrid version of PANORAMA with Horovod we call Horovod hybrid. Just like PANORAMA, Horovod is also equivalent to sequential SGD concerning convergence behavior. Therefore the hybrid of them will remain reproducible.

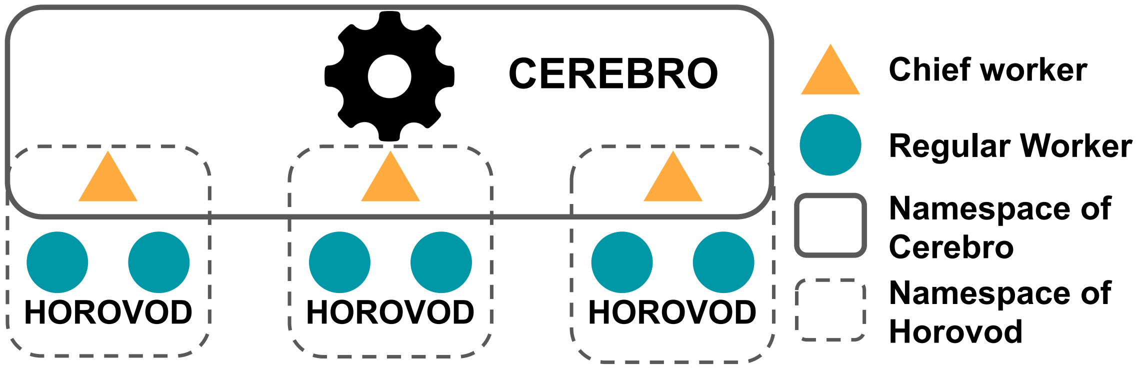


Figure A.3. The architecture of Horovod Hybrid. Within different namespaces, we run PANORAMA and Horovod, respectively. The chief workers, acting as PANORAMA workers, are responsible for driving Horovod tasks and handling the communication between the two systems. In the figure, we show a 9-node cluster with 3 model configs to train.

Figure A.3 summarizes the architecture of Horovod Hybrid, where instead of workers, we have worker groups. Inside each worker group, we run a data-parallel Horovod task. Then after each worker group finishes their assigned task, we hop the trained models just as the regular PANORAMA. We assume there are more workers than model configs. We create an equal number of groups for the number of configs. Workers are placed into these groups evenly.

We now explore the possibility of hybridizing MOP with Horovod to better utilizer resources in $|S| < p$ regime. For this we run an experiment using *Criteo* on the CPU cluster with varying number of configs ($|S|$) and batch sizes. We compare three different methods: (1) Horovod, (2) MOP, and (3) Horovod Hybrid. Figure A.4 shows the results.

Horovod's runtime grows linearly with more configs, but PANORAMA is constant. For instance at batch size 128 and $|S| = 2$, PANORAMA matches Horovod's performance. This is

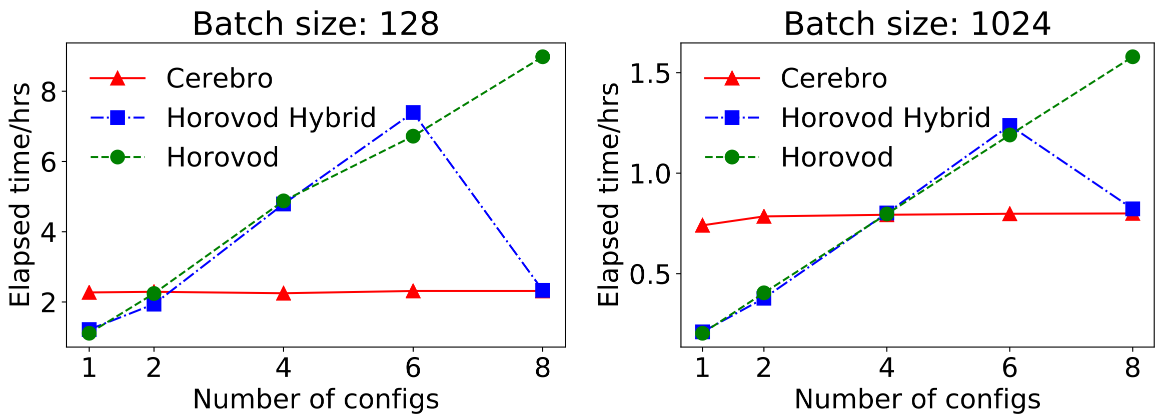


Figure A.4. Performance tests of Horovod Hybrid with varying batch size and $|S|$ on 8-node cluster. Configs: same model as in Section 4 Table 3.5, learning rates drawn from $\{10^{-3}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$, weight decays drawn from $\{10^{-4}, 10^{-5}\}$. We test on 2 different batch sizes, respectively.

because PANORAMA's communication costs are negligible. For the Horovod Hybrid the runtimes are comparable to Horovod, except when $|S| = p$ it reduces to PANORAMA. This is because Horovod Hybrid is bottlenecked by Horovod's network overheads; mitigating this issue will require careful data re-partitioning, which we leave to future work. It is interesting that even with underutilization PANORAMA can still outperform Horovod in most scenarios. Depending on $|S|$ and batch size, there is a cross-over point when the three methods meet. Typically when $|S| \ll p$, Horovod and Horovod Hybrid are faster as PANORAMA is heavily underutilizing the workers. We heuristically choose $p/2$ as the dividing point: still run PANORAMA if $p/2 < |S|$, otherwise just run Horovod. Overall, the current Horovod Hybrid does not provide much benefit over Horovod as it mainly optimizes Horovod for its latency part of the communication cost, which turns out to be marginal.

Table A.1. Computation costs of the CNNs used for the simulation experiment comparing different scheduling methods.

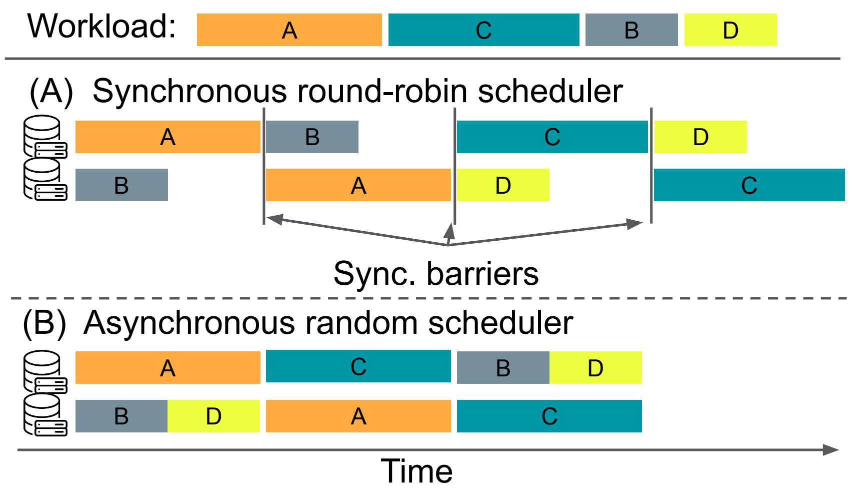
Model	FLOPs
AlexNet	727 MFLOPs
CaffeNet	724 MFLOPs
SqueezeNet1-0	837 MFLOPs
SqueezeNet1-1	360 MFLOPs
VGG-f	727 MFLOPs
VGG-m	2 GFLOPs
VGG-s	3 GFLOPs
VGG-m-2048	2 GFLOPs
VGG-m-1024	2 GFLOPs
VGG-m-128	2 GFLOPs
VGG-vd-16-atrous	16 GFLOPs
VGG-vd-16	16 GFLOPs
VGG-vd-19	20 GFLOPs
GoogleNet	2 GFLOPs
ResNet18	2 GFLOPs
ResNet34	4 GFLOPs
ResNet50	4 GFLOPs
ResNet101	8 GFLOPs
ResNet152	11 GFLOPs
ResNext-50-32x4d	4 GFLOPs
ResNext-101-32x4d	8 GFLOPs
ResNext-101-64x4d	16 GFLOPs
Inception-V3	6 GFLOPs
SE-ResNet-50	4 GFLOPs
SE-ResNet-101	8 GFLOPs
SE-ResNet-152	11 GFLOPs
SE-ResNeXt-50-32x4d	4 GFLOPs
SE-ResNeXt-101-32x4d	8 GFLOPs
SENet	21 GFLOPs
SE-BN-Inception	2 GFLOPs
DenseNet121	3 GFLOPs
DenseNet161	8 GFLOPs
DenseNet169	3 GFLOPs
DenseNet201	4 GFLOPs
MobileNet	579 MFLOPs

Appendix B

CEREBRO on Data Systems

B.1 Scenarios that Could Affect Scheduler Performance

Various scenarios could affect MOP schedulers' performance. One such case is heterogeneous workload, where the sync. and async. schedulers may yield schedules that deviate in makespan. One such scenario is shown in Figure B.1.



Workload:
A
C
B
D
(A) Synchronous round-robin scheduler
A
B
C
D
B
A
D
C
Sync. barriers
(B) Asynchronous random scheduler
A
C
B
D
B
D
A
C
Time

Figure B.1. Gantt chart for possible schedules generated by (A) Synchronous round-robin scheduler and (B) Asynchronous random scheduler. The two workers are homogenous, but the workload, containing four models, is heterogenous.

The other notable example is when the number of model configs approximates the available degree of parallelism as shown in Figure B.2; in this scenario, the async. random scheduler could suffer from straggler issues while the sync. scheduler is free of such problems.

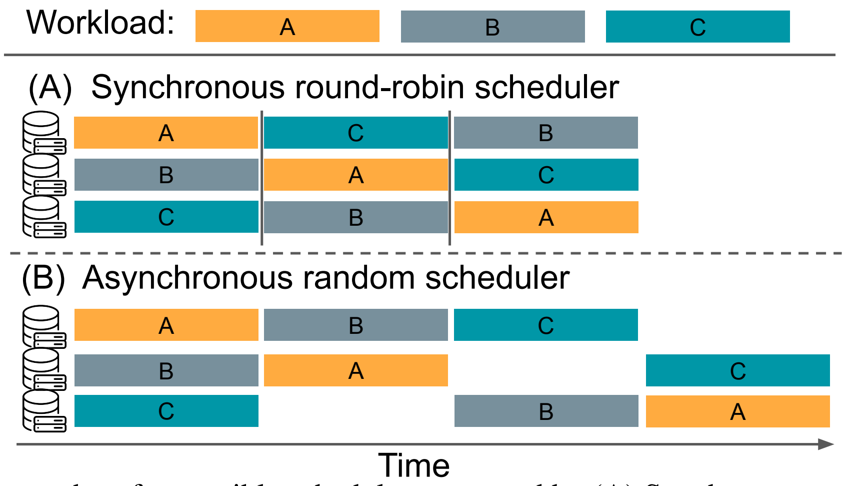


Figure B.2. Gantt chart for possible schedules generated by (A) Synchronous round-robin scheduler and (B) Asynchronous random scheduler.

B.2 Proofs to Propositions

Proof to Proposition 1. For sync. MOP, there will be in total M sub-epoch batches, and each batch's runtime will be dominated by the longest running model within this batch, therefore we have in expectation:

$$T_u = \sum_i^M \max(\mathbb{L}_i), \tag{B.1}$$

where $\mathbb{L}_i = \text{rand}(\mathbb{L}, W)$. $\text{rand}(\mathbb{X}, N)$ means randomly sampling N elements from set $\mathbb{X}$. Assuming that $\max(\mathbb{L}_i) \sim l_s$, meaning every sub-epoch of UDAF run is dominated by l_s, we obtain:

$$T_u = p^W l_s M + (1 - p^W) l_m M. \tag{B.2}$$

On the other hand with async. MOP, assuming M is large enough ($M \gg W$) and the scheduler was capable of load-balancing, we have:

$$T_c = \frac{|\mathbb{L}|}{W} W = W \frac{M\bar{l}}{W} = M\bar{l}. \tag{B.3}$$

Immediately we have the speedup of async. MOP over sync. MOP:

$$\frac{T_u}{T_c} = \frac{p^W l_s M + (1-p^W) l_m M}{M\bar{l}} = p^W \frac{l_s}{\bar{l}} + (1-p^W)\frac{l_m}{\bar{l}}. \tag{B.4}$$

This indicates the speedup (weak scaling) of async. MOP over sync. MOP is related to the number of workers W and the skewness (represented by $\frac{l_s}{\bar{l}}$ and $\frac{l_m}{\bar{l}}$). Interestingly, note that this speedup is independent of the number of model configs.

Proof to Proposition 2. Asymptotically, when W goes up, we can see $\frac{T_u}{T_c} \to \frac{l_m}{\bar{l}}$. Define $\eta =: \frac{l_m}{\bar{l}}$. $\eta > 1$ as long as the underlying distribution of $\mathbb{L}$ is right-tailed. This means async. MOP will always be faster than sync. MOP under such circumstance, given sufficient number of workers.

Furthermore, since $\bar{l} = \frac{pl_s M + (1-p) l_m M}{M} = pl_s + (1-p)l_m$, if we expand η, there is:

$$\eta = \frac{l_m}{\bar{l}} = \frac{l_m}{pl_s + (1-p)l_m}. \tag{B.5}$$

When $p \to 1$ we obtain

$$\eta \to \frac{l_m}{l_s}, \tag{B.6}$$

which is unbounded and can potentially go to a very high number under extreme circumstances. This indicates when there are only a few outlier models that are time-consuming, the speed-up of async. MOP over sync. MOP is determined by the relative runtime difference of the outlier models and common models.

B.3 Effect of Model Size on UDAF and CTQ

The size of models is typically orders of magnitude smaller than the size of the training dataset. Thus, although model hopping time is proportional to model size, it is usually negligible in large-scale DL. However, this assumption may not hold for the UDAF approach because of the `JOIN` as explained in Section 4.5.1 Model hopping. We run the following test to investigate model transmission cost with varying model sizes empirically.

We choose 3 hyperparameter tuning workloads on ImageNet. Each workload features 8 model configs with one single model architecture and a fixed batch size of 32. Model architectures are: MobileNet (52 MB), ResNet50 (294 MB), and ResNet152 (693 MB). Model size is reported as the on-disk serialized size. We run each workload for 3 epochs and take the average to get per epoch and per worker machine time breakdown.

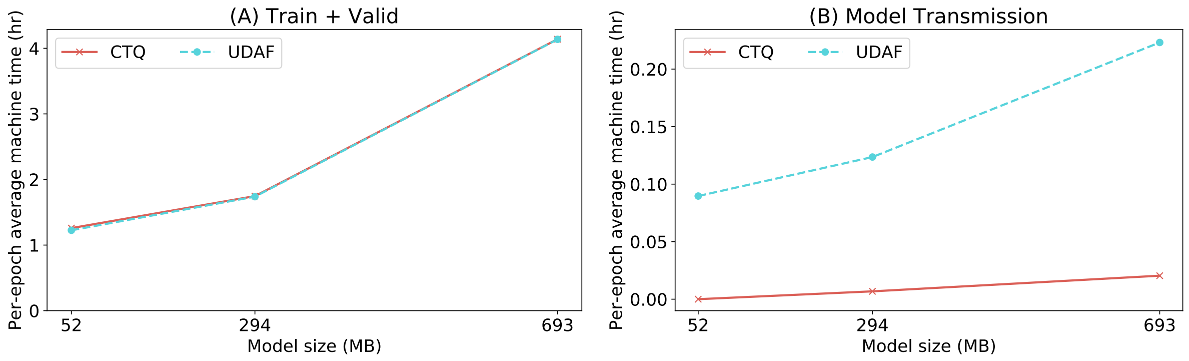


Figure B.3. Per-epoch runtime for model size test. (A): Train+Valid time. (B): Model Transmission time.

Figure B.3 presents the results. We profile and focus on two components: Train+Valid and Model Transmission. Figure B.3(A) shows no difference in terms of Train+Valid, which is to be expected. Figure B.3(B) shows that the CTQ approach imposes little to no bottleneck and is far less sensitive to the model size. However, the UDAF approach suffers more overheads on larger models. This confirms that the JOIN and storing models inside the DB can indeed cause some overheads, although this overhead is not too major (less than 10% in this case).

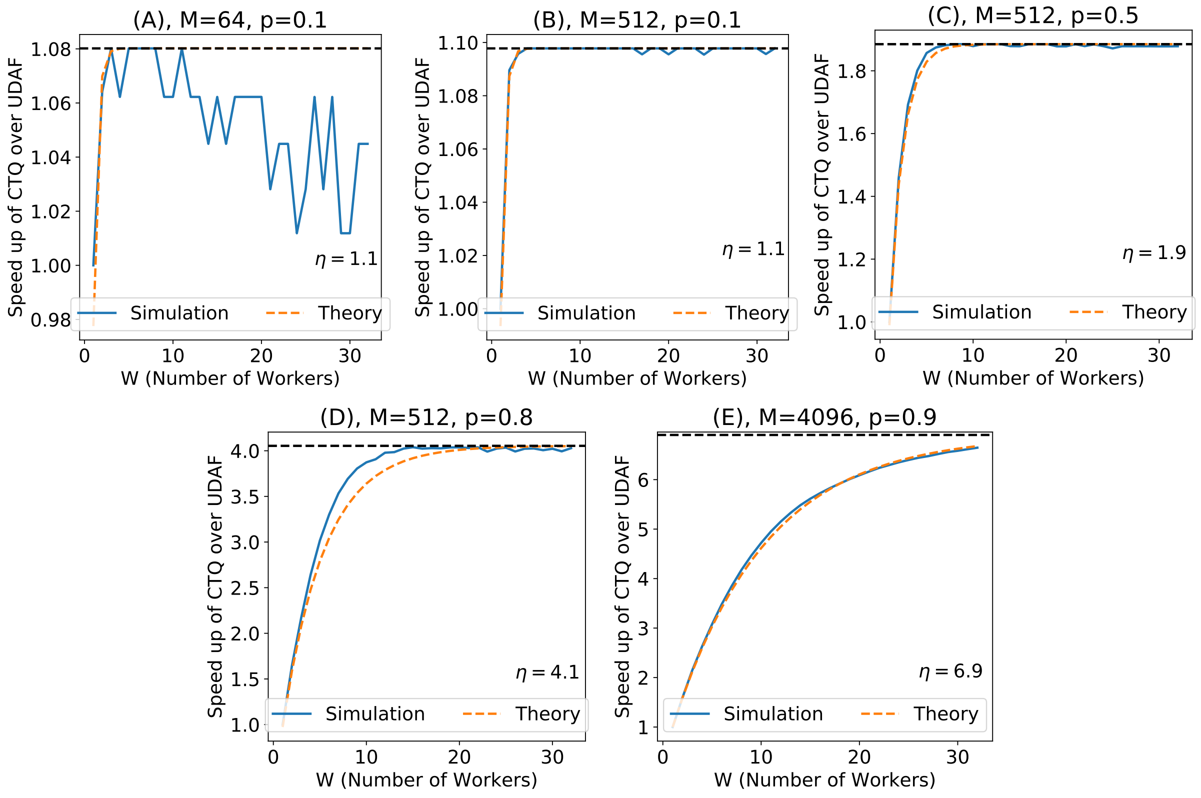


Figure B.4. Runtimes of heterogeneous workloads. (A-E) represent different workload configs. Both theoretical bounds and simulated runtime gaps are shown. The upper bounds of speedup η are calculated for each workload. NB: Note the different ranges of the Y axes across plots.

B.4 Simulated Extreme Scenarios of Async. MOP vs Sync. MOP on Heterogeneous Workloads

To supplement the experiments shown in Section 4.6.2, we now add more simulated tests to show both theoretical and simulated performance gain of Async. MOP over Sync. MOP on heterogeneous workloads. We assume $l_m/l_s = 20$. We evaluate the speedups of CTQ over UDAF for different numbers of workers W (up to 32). Figure B.4 presents the results. Comparing Figure B.4(A) with (B), we see that when M is not large enough, the theoretical upper bound is too loose as it assumes a sufficient number of models for the MOP scheduler to load-balance. Comparing Figure B.4(B) with (C), we see when p goes up, which means higher heterogeneity and higher right-skewness, CTQ offers drastically higher speedups over UDAF. Furthermore,

Table B.1. Resource utilizations from Hyperopt experiments shown in Section 4.6.2.

Approach	GPU util.	GPU RAM util.	CPU util.	DRAM util.	Total Network	Per Worker Disk R/W
UDAF	32.5%	16.2%	2.4%	7.2%	600 GB	12 GB / 173 GB
CTQ	33.2%	16.5%	2.5%	1.5%	600 GB	12 GB / 92 GB
DA-Cerebro	45.3%	24.1%	2.0%	20.2%	500 GB	0.7 GB / 6.7 GB
Cerebro-Spark	43.6%	24.2%	13.4%	15.8%	1000 GB	0.3 GB / 2 GB
Hyperopt-Spark	44.6%	24.0%	4.2%	3.6%	20 GB	3000 GB / 4 GB

as Figure B.4(E) shows, CTQ's performance gain can continue to increase with even higher skewness η.

B.5 Hyperopt Experiment Resource Utilizations

Table B.1 summarizes the detailed resource utilization figures for experiments shown in Section 4.6.2. Hyperopt-Spark is task parallel. Therefore it has little network usage compared to other approaches. It requires full data replication; data is not partitioned, and each worker keeps an entire copy of the whole dataset. Because the full dataset does not fit in the DRAM of a single node, it does not cache the data and resorts to frequent disk reads, resulting in orders of magnitude higher disk reads. Despite the MOP-based methods have higher network usage and Hyperopt-Spark has higher disk reads, these are still minor overheads, and none of the approaches is bounded by network nor disk R/W.

B.6 End-to-end Tests with PyTorch Lightning

As the DL training scheme continues shifting after the publishment of this paper, we further include this experiment against PyTorch Lightning DDP, an emerging and popular choice for data parallel model training. We use PyTorch 2.0.1 and PyTorch Lightning version 2.0.6 (both are the latest stable release when this section is written) on CUDA 11.7. We take the same ImageNet end-to-end test from Section 4.6.1 and run it on the same set of hardware so that the results can be directly comparable. Figure B.5 shows the convergence behavior, and Table B.2 lists the runtime and resource utilization measurements. Overall, there is no noticeable

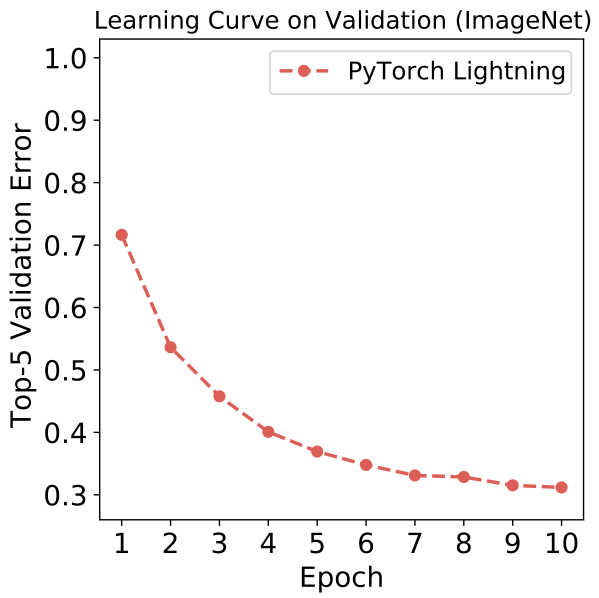


Figure B.5. Convergence behavior on ImageNet.

difference between PyTorch Lightning (which uses DDP under the hood) and plain PyTorch DDP shown in Section 4.6.1, even after the software and CUDA version updates. All the numbers, including the convergence behavior, are almost identical. The fundamental challenge of data parallel training, network communication, has not been mitigated by the recent iteration of these frameworks. We still see network traffic of over 2000 TB. During this test, we did not cache the training data in memory due to technical difficulties with Lightning, which resulted in a higher disk Read than PyTorch DDP. Lightning also employs automatic checkpointing, which is likely the cause of the higher disk Write. This test also shows higher GPU RAM usage than before, which might be due to changed behavior for data pre-fetching/GC policy in the newer versions of PyTorch/Lightning. That being said, these are all minute details as the disk R/W and GPU RAM are not the bottleneck and have little effect on the end-to-end results. Overall, this test does not change any of our conclusions reported in Section 4.6.1.

Table B.2. Runtimes and resource utilizations of end-to-end tests. Execution time and all utilizations are measured, excluding ETL. Per-epoch time equals Execution time divided by the number of epochs. Total network means the total amount of data transmitted during execution. We report disk read/write as per worker average.

Approach	ETL time	Exec. time	Epoch time	GPU util.	GPU RAM util.	CPU util.	DRAM util.	Tol. network	Per w. disk R/W
Lightning	4.4 hr	77.7 hr	7.7 hr	97.3%	41.8%	7.0%	18.0%	2016 TB	392 GB / 47 GB

Appendix C

Lotan

C.1 Appendix

C.1.1 Cost Models

Replication Factor

The vertex replication factor is a common measure of the quality of graph partitioning algorithms. It is defined as the average amount of logical presence each vertex has across partitions. However, replication factor defined this way does not consider the asymmetry during GNN forward- and backward-propagation highlighted in Section 5.5.1.

During forward-prop, the vertices merely have their embeddings, and replicating these vertices is relatively less expensive. However, during back-prop, the vertices are associated with maps of gradients that could be orders of magnitude larger than the embeddings. Replicating them would induce high costs. Hence vertex replication has different importance during forward-prop and back-prop. To account for this asymmetry, we define, as follows, a new metric composed of a weighted sum of the forward and backward replication costs.

Define the set of vertex partitions $\mathbb{V}_p = \{(v_i, p)\}$, each vertex v_i is accompanied by the partition number p. If a vertex is replicated, multiple tuples will be in the set with the same vertex but different partitions. Similarly we define the set of edge partitions as $\mathbb{E}_p = \{(v_i, v_j, p)\}$, where v_i is the source and v_j the destination.

Forward replication factor. During forward-prop, data flows from source vertices to destination vertices. Replication, if needed, happens during the scatter phase when the source and destination are not colocated. The same source vertex needs to be shipped over the network to each physical location where it is needed. Hence higher replication factor directly contributes to more networking needed. Define the forward replication cost R_f to be:

$$R_f := \frac{1}{n}\sum_i |\mathbb{A}_f(v_i)|, \tag{C.1}$$

where $\mathbb{A}_f(v_i) \subseteq \{p\}$ is the subset of partitions that v_i is mirrored to, and v_i has at least one outgoing edge that is co-located in that partition.

Backward replication cost. Similarly, the backward replication cost R_b can be defined as:

$$R_b := \frac{1}{n}\sum_i |\mathbb{A}_b(v_i)|, \tag{C.2}$$

where $\mathbb{A}_b$ is defined the same way as $\mathbb{A}_f$, except that instead of summing all v_i that have an out-going edge, we now sum those that have an in-coming one.

Total replication cost. Together, we take a weighted sum of R_f and R_b to obtain the total replication cost R:

$$R := \frac{1}{1+d}R_f + \frac{d}{1+d}R_b, \tag{C.3}$$

where d is the average degree of the graph. R_f and R_b now acknowledge the asymmetry between forward- and back-propagation. In practice, R_f and R_b can be measured using their definitions rather easily.

Memory Consumption

The most intensive memory consumption of the Graph Engine comes from the gather-scatter operations. We can model the relationship between memory consumption and the number of partitions.

$$M = \frac{f_{rep}P + N}{\max(\frac{P}{ML}, 1)}, \tag{C.4}$$

where M is the number of machines, L is the number of processing units per machine (degree of parallelism), P is the number of partitions for the data, and N is the total amount of data (in terms of the number of vertices). For simplicity, assume f_{rep} follows a linear relationship with P. Observations:

1. At the very low amount of partitions ($P \leq ML$), increase P would increase memory footprint.

2. When $P > ML$, the memory consumption would eventually begin to drop.

This means the memory consumption would rise and then fall as P increases.

Overheads

It is a non-trivial task to manage and operate on billions of objects in a distributed environment. Overheads such as object headers and one extra ephemeral copy of data are negligible in many systems. However, we cannot safely ignore them due to the "amplifying" effect of large graphs – any inefficiency would get repeated millions, if not billions of times, due to the number of vertices and edges in such a graph. Consequently, we realized our cost model has to have a better understanding and estimation of the overheads for a more accurate total cost estimation. We separate the overheads into two categories: constant and scaling. As the name suggests, the constant overheads are fixed costs associated with each specific type of operation;

these could include process setup/destruction time. They usually do not scale with the amount of data and, therefore of little importance to our estimation.

On the other hand, the scaling overheads rise when the number of data increases. Note that the trend may not always be linear, as when the data scale approaches certain thresholds (disk/network throughput, RAM constrain), new overheads are induced due to network throttling, disk spilling, and garbage collections. All in all, these overheads further complicate the picture and are even harder to estimate. Our system relies on logs of past runs and specific heuristics to determine their costs.

Induced Overheads. Given the total memory consumption I, network and disk bandwidth usage J and K. And their respective resource limit $I_{max}, J_{max}, K_{max}$. The total induced overhead $O_{induced}$ can be summarized as:

$$O_{induced} := O_{memory} + O_{network} + O_{disk}, \tag{C.5}$$

and

$$O_{memory} = \mathbf{1}_{I>I_{max}} \cdot o_{memory}(I), \tag{C.6}$$

$$O_{network} = \mathbf{1}_{J>J_{max}} \cdot o_{network}(J), \tag{C.7}$$

$$O_{disk} = \mathbf{1}_{K>K_{max}} \cdot o_{disk}(K), \tag{C.8}$$

where $o_{memory}, o_{network}, o_{disk}$ are the respective functions for the overheads, and $\mathbf{1}_A$ is the indicator function defined as:

$$\mathbf{1}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \tag{C.9}$$

This means the induced overheads only exist when the required resource is above the resource limit and also scales along with the amount of resource requirement. There is no good way to estimate these functions and the resource limits beforehand, and we usually rely on runtime statistics. Even so, the estimation may still be tricky due to its non-linear nature. Hence a quick workaround is to give preference to execution plans with resource requirements below thresholds, which is implemented in Lotan.

Computational Cost Models

Scatter-Gather-Collect Cost.

$$W_{SGC}(k, forward) = l(k)N \cdot (c_0 d_{out} + c_1 R_f + c_2 d_{in}) \tag{C.10}$$

The first term is the scatter computation time, the second is the scatter data movement time (involving network and disk I/O), and the third is the gather and collect computation time. c_0, c_1, c_2 represent the throughput of scatter, data movement and gather, respectively. Similarly, for back-propagation:

$$W_{SGC}(k, backward) = l(k)N \cdot (c_0 d_{in} + c_1 R_b f_e + c_2 d_{out}). \tag{C.11}$$

Note the asymmetry between forward and backward propagation, as highlighted previously in Section 5.6. Furthermore, we define an explosion factor $f_e \in \{d_{in}, d_{out}, 1\}$ to represent the potential explosion of data for specific plans. The actual value of f_e depends on the specification of the GNN, the execution plan, and the current prorogation direction.

Pipe-and-Join Cost.

$$W_{PJ}(k) = c_3 l(k)N(f_e^{out} + f_e^{in}) + c_4 N + c5l(k)N, \tag{C.12}$$

The first and second terms (collected together) represent the pipe-to and pipe-from cost between the Graph Engine and Deep Learning Engine. Hence, to indicate the potential asymmetry, we have two explosion factors f_e^{out} and f_e^{in}. The third term represents the join cost of adding data back to the Graph Engine, and we always use a hash-join. The last term is the serialization/deserialization costs between different runtimes. c_3, c_4, c_5 are the pipe throughput, join operator, and serialization coefficients, respectively.

ApplyEdge-Aggregation-ApplyVertex Cost.

$$W_{AAA} = l(k) N f_e (w_0(k) + w_1(k)) + w_2(k) l(k) N, \tag{C.13}$$

where w_0, w_1, w_2 are the speed of the ApplyEdge, Aggregation, and ApplyVertex functions; they all depend on the GNN model specification.

Total Cost. To put everything together, we have the total cost of an execution plan written as:

$$W = \sum_k \sum_{direction} (W_{SGC} + W_{PJ} + W_{AAA} + O(N, d, l)), \tag{C.14}$$

where $O(N, d, l)$ is the non-negligible overheads associated with each stage. To compute the total cost, we need to gather statistics or estimate all the coefficients, compute the costs for each stage, and then sum them together.

C.1.2 Messenger

The architecture of the Messenger is shown in Figure C.1. We create one Dealer per Graph Engine worker to handle the datacasting and batching, dubbed micro-batch processing. Each data batch is hash-indexed to verify the data orders. The Dealers then connect to a Router, which forwards data to and from the message queues, which the DL Engine workers consume.

C.1.3 Supplementray Experiment Results

Learning Curves

Figure C.3 shows the learning curves for the best model out of some of the hyperparameter tuning workloads on the validation set.

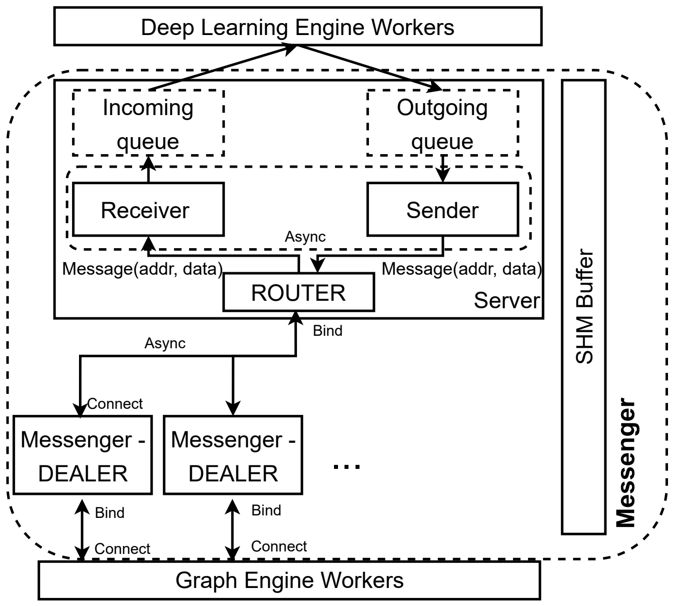
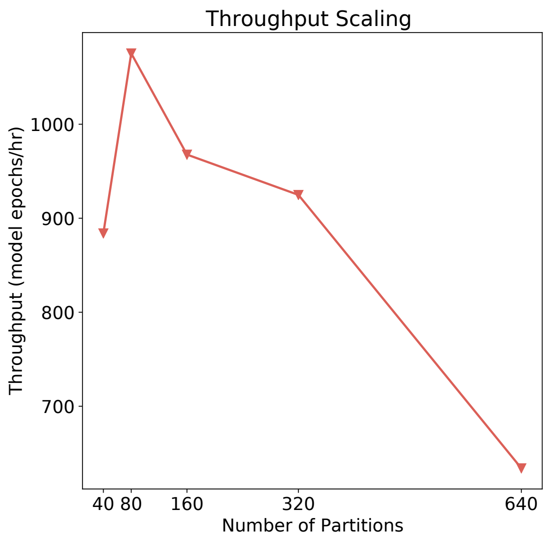


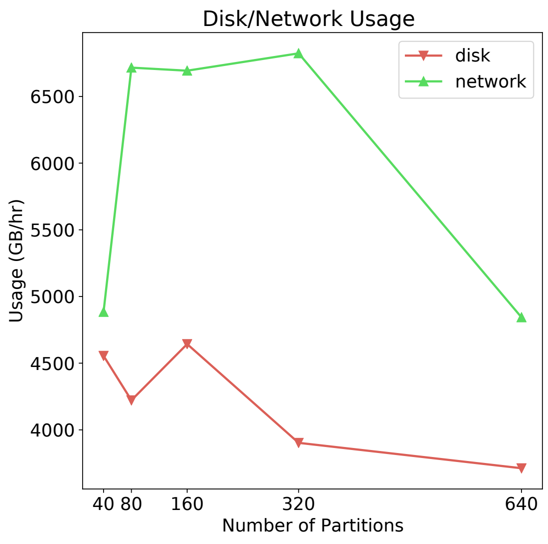
Figure C.1. Messenger architecture.

Effect of number of partitions

In our system, the number of partitions is a critical config parameter to tune. This parameter interplays with various components and subtly impacts the overall performance. We take the same ogbn-arxiv+GCN workload used in our end-to-end experiments and run it with various partitions. Figure 5.10 (C) shows the experiment results. As predicted in Section 5.6, the throughput first increases and then decreases. It increases likely due to increased parallelism at the beginning but then drops because of the overheads caused by the higher number of partitions. The network usage follows a similar trend, but the disk usage remains more stable. It is important to note that there exists a sweet spot of the parameter setting for maximum throughput. However, as discussed earlier, it is tough to model such non-linear behavior. So instead, we adopt the heuristics described in Section 5.6.



(a) Throughput.



(b) Disk/Network Usage.

Figure C.2. Effect of the number of partitions.

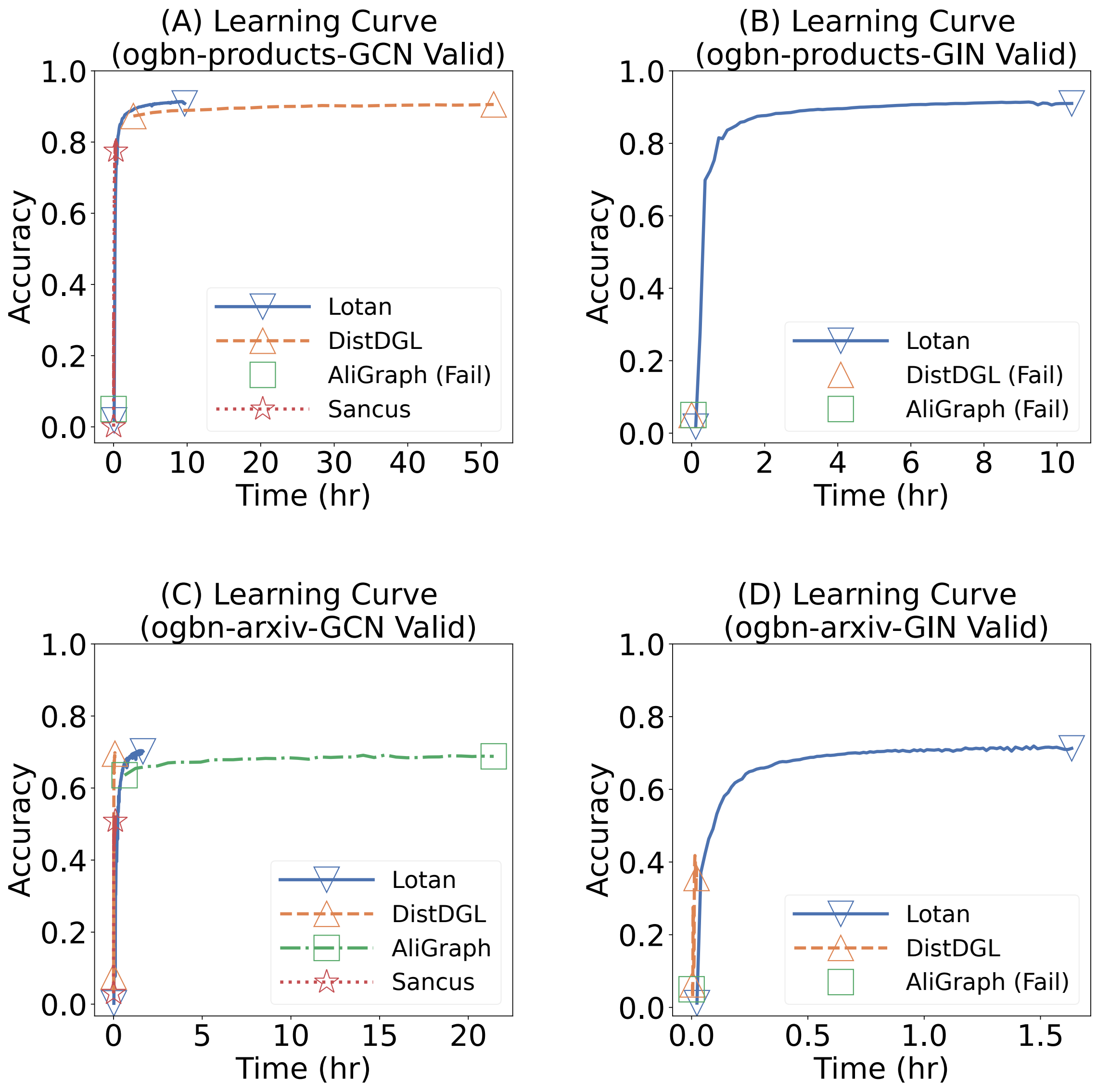


Figure C.3. Learning curves for the chosen model on the validation set. (A) ogbn-products-GCN. (B) ogbn-products-GIN. (C) ogbn-arxiv-GCN. (D) ogbn-arxiv-GIN.

Appendix D

Panorama

D.1 Stem$_1$

Figure D.1 shows Stem$_1$.

D.2 YOLOv2 Loss

The Yolov2 loss is:

$$\begin{aligned} l_k = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\ + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\ + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\ + \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 . \end{aligned} \quad \text{(D.1)}$$

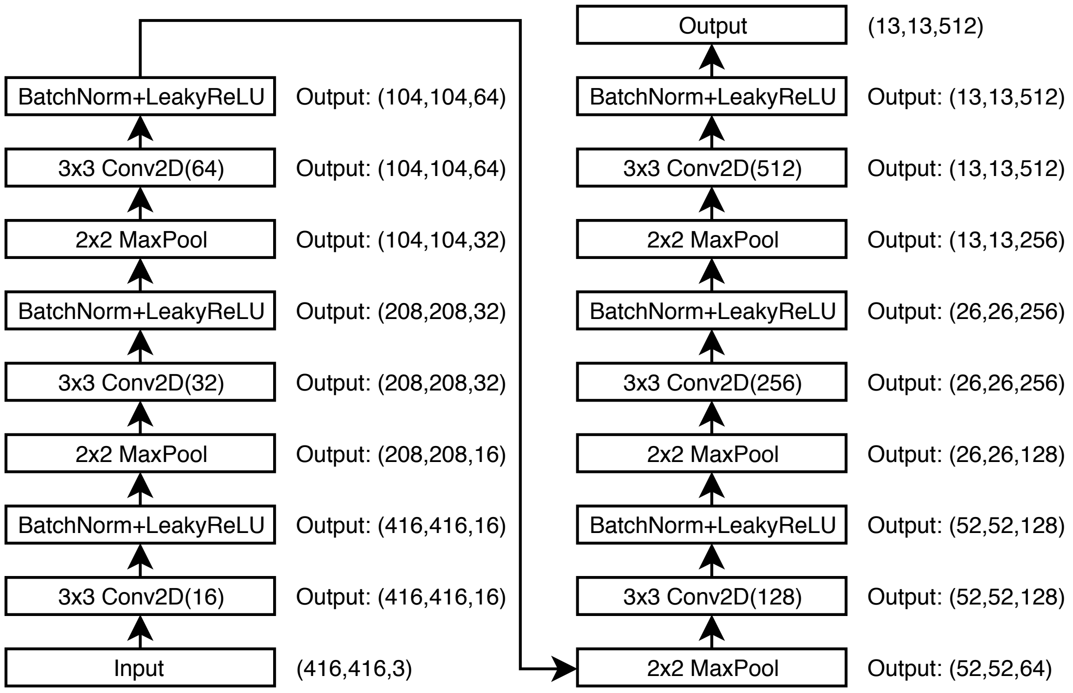


Figure D.1. Detailed architecture of Stem$_1$ block from Figure 6.7. Max pooling layers have a stride of 2 and are valid-padded; other layers have a stride of 1 and are same-padded.

D.3 Responsible AI: First Step

Responsible AI has become one of the most critical topics in today's AI research community. We believe fairness, accountability, transparency, and ethics are prerequisites for any AI system that make predictions about people. It is the community's responsibility in building and testing systems with these prerequisites in mind. Recent proposals like Model Card [203] provides a good template for researchers about how to approach responsible use of AI.

Although Panorama is a domain-agnostic and not a face/person recognition system, it can be used for such uses. Therefore we believe it is necessary to test its behavior regarding fairness and show what the limitations are. Following the practice of Gender Shades [57], we decide to test Panorama's performance on different gender-shades groups of people. We then try to report as many details as we could. However, please note these tests are only the first step, and

	*LM	DF	DM	LF
Ref. model	90.0%	88.2%	79.8%	78.4%
PanoramaNet	59.0%	48.4%	44.2%	43.8%
Random guessing	0.3%	0.3%	0.3%	0.3%

Table D.1. Reference model and Panorama's performance on different gender-shades groups. *D: `darker`, L: `lighter`, F: `female`, M: `male`.

our report may not comply with the high standard of Model Card, because most datasets we used lack the needed information about gender-shades.

Definitions of Gender and Shades. We are aware that the definition of gender and shades are extremely nuanced, and there cannot be a unified standard. We mostly follow prior research [57] by defining shades to be binary `darker` and `lighter` based on the Fitzpatrick Skin Type. Gender is the oversimplified perceived sex label of `male` and `female`.

Task. We take the out-vocabulary recognition task as an example and reuse the reference model and PanoramaNet in Section 6.5.2.

Datasets. The reference model was trained on VGGFaces, WIDER FACE, CelebA datasets; PanoramaNet was trained on the reference-model-labeled CBSN dataset. The gender-shades information of these datasets is unknown. Based on the four gender-shades groups of {`lighter, darker`}-{`female, male`}, we manually labeled 25 identities from Youtube-Faces for each group. In total, we have 100 identities. We test both the reference model and PanoramaNet on this sub-sampled dataset.

Metrics. We use the same metric as in Section 6.5.2.

Queries and results. We sample 20 queries for each identity, resulting in 2000 queries. We then report the results on a per-group basis. Table D.1 summarizes the quantitive results.

Both models show performance differences in different groups and may be biased. We observe that Panorama mostly copied the reference model; the accuracy ranks as $LM > DF > DM > LF$ for both models. However, we also notice that some gap is amplified; $LM - DF$

differences rise from 1.8% to 9.6%. Meanwhile, some gap is damped; $DF - DM$ decreases from 8.4% to 4.2%. This indicates that the reference model may not be the only source of bias; the video stream could be one of the other major factors.

In conclusion, Panorama's fairness behavior will be determined by the reference model and the video stream's fairness behavior. To enforce fairness, it would require at least: either the user to make sure the inputs are not biased; Or the system has to have a built-in notion of gender-shades, so that it can make sure the training dataset generated from user's reference model and video stream is balanced, by sampling based on these notions. However, it is still unknown to us how to build and integrate such functionalities into the system. There are also open questions regarding why there are relative performance differences in different groups from an unbound vocabulary point of view; Why it is harder to generalize to certain groups. We leave all of these topics for future discussions.

Bibliography

[1] Cerebro Documentation. https://adalabucsd.github.io/cerebro-system/.

[2] First hand knowledge from the authors.

[3] NPR: Facial Recognition In China Is Big Business As Local Governments Boost Surveillance. https://www.npr.org/sections/parallels/2018/04/03/598012923/facial-recognition-in-china-is-big-business-as-local-governments-boost-surveilla, 2018. [Online; accessed 30-September-2023].

[4] The Economic Times: Computer Vision for Crowd Control at India's Kumbh Mela. https://economictimes.indiatimes.com/news/politics-and-nation/higher-budget-and-bigger-ground-this-years-kumbh-mela-is-set-to-begin-with-a-bang/articleshow/67397579.cms, 2019. [Online; accessed 30-September-2023].

[5] DeepPostures. https://adalabucsd.github.io/DeepPostures/, 2023. [Online; accessed 30-September-2023].

[6] About Greenplum Query Processing, Accessed 30-September-2023. https://gpdb.docs.pivotal.io/560/admin_guide/query/topics/parallel-proc.html.

[7] Code Release of This Work, Accessed 30-September-2023. https://github.com/makemebitter/cerebro-ds.

[8] Create, Train, and Deploy Machine Learning Models in Amazon Redshift Using SQL with Amazon Redshift ML, Accessed 30-September-2023. https://shorturl.at/bvBE8.

[9] Google BigQuery ML, Accessed 30-September-2023. https://cloud.google.com/bigquery-ml/docs.

[10] Google BigQuery ML TensorFlow integration, Accessed 30-September-2023. https://cloud.google.com/bigquery-ml/docs/making-predictions-with-imported-tensorflow-models.

[11] Horovod on Spark, Accessed 30-September-2023. https://github.com/horovod/horovod/blob/master/docs/spark.rst.

[12] MADlib Deep Learning, Accessed 30-September-2023. https://madlib.apache.org/docs/latest/group__grp__dl.html.

[13] MADlib Model Selection, Accessed 30-September-2023. https://madlib.apache.org/docs/latest/group__grp__mdl.html.

[14] Microsoft SQL Server Machine Learning Services, Accessed 30-September-2023. https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017.

[15] Oracle Data Mining, Accessed 30-September-2023. https://www.oracle.com/database/technologies/advanced-analytics/odm.html.

[16] Oracle Machine Learning, Accessed 30-September-2023. https://www.oracle.com/data-science/machine-learning.html.

[17] Script for Tensorflow Model Averaging, Accessed 30-September-2023. https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/utils/avg_checkpoints.py.

[18] TensorFrames, Accessed 30-September-2023. https://github.com/databricks/tensorframes.

[19] The CREATE MODEL Statement for Deep Neural Network (DNN) Models, Accessed 30-September-2023. https://cloud.google.com/bigquery-ml/docs/reference/standard-sql/bigqueryml-syntax-create-dnn-models.

[20] TOAST Tables in Postgres, Accessed 30-September-2023. https://wiki.postgresql.org/wiki/TOAST.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283. USENIX Association, 2016.

[22] D. A. Adjeroh and K. C. Nwosu. Multimedia database management requirements and issues. In *IEEE MultiMedia*, volume 4, pages 24–33, 1997.

[23] 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 *CIDR*. www.cidrdb.org, 2020.

[24] Determined AI. AI Infrastructure for Everyone, Now Open Source, Accessed 30-September-2023. https://determined.ai/blog/ai-infrastructure-for-everyone/.

[25] Z. Akata, F. Perronnin, Z. Harchaoui, and C. Schmid. Label embedding for attribute-based classification. In *CVPR*, 2013.

[26] Ryo Akita, Akira Yoshihara, Takashi Matsubara, and Kuniaki Uehara. Deep learning for stock prediction using numerical and textual information. In *ICIS*, pages 1–6. IEEE Computer Society, 2016.

[27] Samuel Albanie. Memory Consumption and FLOP Count Estimates for Convnets, Accessed 30-September-2023. https://github.com/albanie/convnet-burden.

[28] Amazon. RedShift Query Planning and Execution Workflow, Accessed 30-September-2023. https://docs.aws.amazon.com/redshift/latest/dg/c-query-planning.html.

[29] Huynh Ngoc Anh. keras-yolo2. https://github.com/experiencor/keras-yolo2, 2018. [Online; accessed 30-September-2023].

[30] Robin Anil, Gökhan Çapan, Isabel Drost-Fromm, Ted Dunning, Ellen Friedman, Trevor Grant, Shannon Quinn, Paritosh Ranjan, Sebastian Schelter, and Özgür Yilmazel. Apache Mahout: Machine Learning on Distributed Dataflow Systems. *J. Mach. Learn. Res.*, 21:127:1–127:6, 2020.

[31] Sabeur Aridhi, Alberto Montresor, and Yannis Velegrakis. BLADYG: A graph processing framework for large dynamic graphs. *Big Data Res.*, 9:9–17, 2017.

[32] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[33] Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The Object-Oriented Database System Manifesto. In *DOOD*, pages 223–240. North-Holland/Elsevier Science Publishers, 1989.

[34] Youhui Bai, Cheng Li, Zhiqi Lin, Yufei Wu, Youshan Miao, Yunxin Liu, and Yinlong Xu. Efficient data loader for fast sampling-based gnn training on large graphs. *IEEE Transactions on Parallel & Distributed Systems*, (01):1–1, 2021.

[35] E. Bart and S. Ullman. Cross-generalization: learning novel classes from a single example by feature replacement. In *CVPR*, 2005.

[36] Ilaria Bartolini and Marco Patella. A general framework for real-time analysis of massive multimedia streams. *Multimedia Systems*, 24(4):391–406, Jul 2018.

[37] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*, pages 1387–1395. ACM, 2017.

[38] A. Bendale and T. Boult. Towards open world recognition. In *CVPR*, 2015.

[39] Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *corr abs/1502.04390*, 2015.

[40] James Bergstra, R. Bardenet, Balázs Kégl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.

[41] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[42] Laure Berti-Équille, Angela Bonifati, and Tova Milo. Machine Learning to Data Management: A Round Trip. In *ICDE*, pages 1735–1738. IEEE Computer Society, 2018.

[43] Dimitri P. Bertsekas. A new class of incremental gradient methods for least squares problems. *SIAM J. on Optimization*, 7(4):913–926, April 1997.

[44] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *CoRR*, abs/1910.09017, 2019.

[45] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.*, 9(13):1425–1436, 2016.

[46] Matthias Boehm, Arun Kumar, and Jun Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[47] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.*, 11(12):1755–1768, August 2018.

[48] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.*, 7(7):553–564, 2014.

[49] Leon Bottou. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proceedings of the Symposium on Learning and Data Science*, 2009.

[50] Xavier Bouthillier and Gaël Varoquaux. Survey of Machine-Learning Experimental Methods at NeurIPS2019 and ICLR2020. Research report, Inria Saclay Ile de France, January 2020.

[51] S. Boyd and L. Vandenberghe. *Convex Optimization*. 2004.

[52] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Process. Mag.*, 34(4):18–42, 2017.

[53] B. Brouwer. YouTube Now Gets Over 400 Hours Of Content Uploaded Every Minute. https://www.tubefilter.com/2015/07/26/youtube-400-hours-content-every-minute/, 2015. [Online; accessed 30-September-2023].

[54] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[55] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag, 3rd edition, 2001.

[56] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016.

[57] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *FAT*, volume 81 of *Proceedings of Machine Learning Research*, pages 77–91. PMLR, 2018.

[58] Z. Cai, M. J. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *ICCV*, pages 3361–3369, 2015.

[59] Z. Cai and N. Vasconcelos. Cascade r-cnn: Delving into high quality object detection. In *CVPR*, 2018.

[60] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[61] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor. Scaling video analytics on constrained edge nodes. In *SysML*, 2019.

[62] CBS. CBS News Live. https://www.cbsnews.com/live/, 2019. [Online; accessed 30-September-2023].

[63] Supriyo Chakraborty, Richard Tomsett, Ramya Raghavendra, Daniel Harborne, Moustafa Alzantot, Federico Cerutti, Mani B. Srivastava, Alun D. Preece, Simon Julier, Raghuveer M. Rao, Troy D. Kelley, Dave Braines, Murat Sensoy, Christopher J. Willis, and Prudhvi Gurram. Interpretability of Deep Learning Models: A Survey of Results. In *SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*, pages 1–6. IEEE, 2017.

[64] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[65] Chun-Fu Richard Chen, Gwo Giun Chris Lee, Yinglong Xia, W Sabrina Lin, Toyotaro Suzumura, and Ching-Yung Lin. Efficient multi-training framework of image deep learning on gpu cluster. In *2015 IEEE International Symposium on Multimedia (ISM)*, pages 489–494. IEEE, 2015.

[66] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 941–949. PMLR, 2018.

[67] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *ICLR (Poster)*. OpenReview.net, 2018.

[68] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 1725–1735. PMLR, 2020.

[69] X. Chen, A. Shrivastava, and A. Gupta. Neil: Extracting visual knowledge from web data. In *ICCV*, pages 1409–1416, 2013.

[70] Yu Cheng, Chengjie Qin, and Florin Rusu. GLADE: big data analytics made easy. In *SIGMOD Conference*, pages 697–700. ACM, 2012.

[71] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. ARDA: Automatic Relational Data Augmentation for Machine Learning. *Proc. VLDB Endow.*, 13(9):1373–1387, 2020.

[72] European Commission. GDPR, Accessed 30-September-2023. https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en.

[73] CriteoLabs. Kaggle Contest Dataset Is Now Available for Academic Use!, Accessed 30-September-2023. https://ailab.criteo.com/category/dataset.

[74] Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge J. Belongie. Large scale fine-grained categorization and domain-specific transfer learning. *CVPR*, pages 4109–4118, 2018.

[75] databricks. Deep Learning Pipelines for Apache Spark, Accessed 30-September-2023. https://github.com/databricks/spark-deep-learning.

[76] Databricks. Introducing Apache Spark 2.4, Accessed 30-September-2023. https://databricks.com/blog/2018/11/08/introducing-apache-spark-2-4.html.

[77] Databricks. Resource-efficient Deep Learning Model Selection on Apache Spark, Accessed 30-September-2023. https://bit.ly/3esN3JT.

[78] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-scale Hierarchical Image Database. In *CVPR*, pages 248–255. IEEE, 2009.

[79] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD Conference*, pages 1701–1716. ACM, 2020.

[80] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[81] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[82] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[83] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. AIDA - Abstraction for Advanced In-Database Analytics. *Proc. VLDB Endow.*, 11(11):1400–1413, 2018.

[84] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed Linear Algebra for Large-Scale Machine Learning. *Proc. VLDB Endow.*, 9(12):960–971, 2016.

[85] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[86] Facebook. Introducing FBLearner Flow: Facebook's AI backbone, Accessed 30-September-2023. https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/.

[87] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. Vertica-ML: Distributed Machine Learning in Vertica Database. In *SIGMOD Conference*, pages 755–768. ACM, 2020.

[88] L. Fei-Fei, R. Fergus, and P. Perona. A bayesian approach to unsupervised one-shot learning of object categories. In *ICCV*, 2003.

[89] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. In *IEEE TPAMI*, 2006.

[90] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for In-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325—-336. Association for Computing Machinery, 2012.

[91] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD Conference*, pages 325–336. ACM, 2012.

[92] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[93] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[94] Tibor Fiala. An Algorithm for the Open-shop Problem. *Mathematics of Operations Research*, 8(1):100–109, 1983.

[95] F. Fleuret and G. Blanchard. Pattern recognition from one example by chopping. In *NIPS*, 2005.

[96] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov. Devise: A deep visual-semantic embedding model. In *NIPS*, 2013.

[97] Y. Fu and L. Sigal. Semi-supervised vocabulary-informed learning. In *CVPR*, 2016.

[98] Swapnil Gandhi and Anand Padmanabha Iyer. P3: distributed deep graph learning at scale. In *OSDI*, pages 551–568. USENIX Association, 2021.

[99] Zekai J. Gao, Niketan Pansare, and Christopher M. Jermaine. Declarative Parameterizations of User-Defined Functions for Large-Scale Machine Learning and Optimization. *IEEE Trans. Knowl. Data Eng.*, 31(11):2079–2092, 2019.

[100] J. V. Gautam, H. B. Prajapati, V. K. Dabhi, and S. Chaudhary. A survey on job scheduling algorithms in big data processing. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–11, March 2015.

[101] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1263–1272. JMLR.org, 2017.

[102] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 2017.

[103] Giraph. Apache Giraph, Accessed 30-September-2023. https://giraph.apache.org/.

[104] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A Service for Black-box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[105] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[106] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613. USENIX Association, 2014.

[107] Teofilo Gonzalez and Sartaj Sahni. Open Shop Scheduling to Minimize Finish Time. *JACM*, 1976.

[108] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. In *MIT press*, 2016.

[109] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. In *MIT press* [108].

[110] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[111] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *IJCAI*, pages 1725–1731. ijcai.org, 2017.

[112] Gurobi. Gurobi Optimization, Accessed 30-September-2023. https://www.gurobi.com.

[113] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[114] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, pages 123–136, 2016.

[115] Sona Hasani, Saravanan Thirumuruganathan, Abolfazl Asudeh, Nick Koudas, and Gautam Das. Efficient Construction of Approximate Ad-Hoc ML models Through Materialization and Reuse. *Proc. VLDB Endow.*, 11(11):1468–1481, 2018.

[116] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, Oct 2017.

[117] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CVPR*, pages 770–778, 2016.

[118] Ruining He and Julian J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517. ACM, 2016.

[119] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[120] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[121] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4), 1998.

[122] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Efficient distribution for deep learning on large graphs. In *MLSys GNNSys Workshop*. mlsys.org, 2021.

[123] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 269–286, 2018.

[124] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In *NeurIPS*, 2020.

[125] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. Multi-scale dense networks for resource efficient image classification. In *International Conference on Learning Representations*, 2018.

[126] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[127] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *Proc. VLDB Endow.*, 11(5):566–579, 2018.

[128] Zengfeng Huang, Shengzhong Zhang, Chong Xi, Tang Liu, and Min Zhou. Scaling up graph neural networks via graph coarsening. In *KDD*, pages 675–684. ACM, 2021.

[129] hyperopt. Scaling out search with Apache Spark, Accessed 30-September-2023. http://hyperopt.github.io/hyperopt/scaleout/spark/.

[130] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *arXiv preprint arXiv:1711.09846*, 2017.

[131] Ashesh Jain, Amir R. Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. In *CVPR*, pages 5308–5317. IEEE Computer Society, 2016.

[132] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *SIGMOD Rec.*, 49(1):43–50, 2020.

[133] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Röhm, and Carsten Binnig. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD Conference*, pages 159–173. ACM, 2020.

[134] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Proceedings of the 10th European Conference on Computer Vision: Part I*, ECCV '08, pages 304–317, Berlin, Heidelberg, 2008. Springer-Verlag.

[135] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[136] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *SYSML 2019*, 2019.

[137] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 463–478, 2017.

[138] Jiawei Jiang, Pin Xiao, Lele Yu, Xiaosen Li, Jiefeng Cheng, Xupeng Miao, Zhipeng Zhang, and Bin Cui. Psgraph: How tencent trains extremely large-scale graphs with spark? In *ICDE*, pages 1549–1557. IEEE, 2020.

[139] Yushi Jing, David Liu, Dmitry Kislyuk, Andrew Zhai, Jiajing Xu, Jeff Donahue, and Sarah Tavel. Visual search at pinterest. In *Proceedings of the 21th ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, KDD '15, pages 1889–1898, New York, NY, USA, 2015. ACM.

[140] Kaggle. State of Data Science and Machine Learning 2019, Accessed 30-September-2023. https://www.kaggle.com/kaggle-survey-2019.

[141] O. Kalipsiz. Multimedia databases. In *IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, 2000.

[142] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. In *VLDB*, volume 10, pages 1586–1597, 2017.

[143] Daniel Kang, Peter Bailis, and Matei A. Zaharia. Blazeit: Fast exploratory video queries using neural networks. *CoRR*, abs/1805.01046, 2018.

[144] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. Extending Relational Query Processing with ML Inference. In *CIDR*. www.cidrdb.org, 2020.

[145] Andrej Karpathy. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35/, 2017. [Online; accessed 30-September-2023].

[146] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-Database Learning with Sparse Tensors. In *PODS*, pages 325–340. ACM, 2018.

[147] Mijung Kim and K. Selçuk Candan. Efficient Static and Dynamic In-Database Tensor Decompositions on Chunk-Based Array Stores. In *CIKM*, pages 969–978. ACM, 2014.

[148] Tae-Young Kim and Sung-Bae Cho. Predicting residential energy consumption using cnn-lstm neural networks. *Energy*, 182:72 – 81, 2019.

[149] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[150] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (Poster)*. OpenReview.net, 2017.

[151] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, volume 54 of *Proceedings of Machine Learning Research*, pages 528–536. PMLR, April 2017.

[152] Seongyun Ko and Wook-Shin Han. Turbograph++: A scalable and fast graph analytics system. In *SIGMOD Conference*, pages 395–410. ACM, 2018.

[153] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *Proc. VLDB Endow.*, 12(11):1399–1412, 2019.

[154] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. BoostClean: Automated Error Detection and Repair for Machine Learning. *CoRR*, abs/1711.01299, 2017.

[155] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *Proc. VLDB Endow.*, 9(12):948–959, 2016.

[156] Kubeflow. Kubeflow, Accessed 30-September-2023. https://www.kubeflow.org/.

[157] Arun Kumar. ML/AI Systems and Applications: Is the SIGMOD/VLDB community losing relevance?, Accessed 30-September-2023. https://wp.sigmod.org/?p=2454.

[158] Arun Kumar, Matthias Boehm, and Jun Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1717–1722. Association for Computing Machinery, 2017.

[159] Arun Kumar, Matthias Boehm, and Jun Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1717–1722. ACM, 2017.

[160] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model Selection Management Systems: the Next Frontier of Advanced Analytics. *SIGMOD Record*, 2016.

[161] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *CIDR*. www.cidrdb.org, 2021.

[162] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.*, 12(11):1553–1567, 2019.

[163] Cornell Lab. Cornell Lab FeederWatch Cam at Sapsucker Woods. http://cams.allaboutbirds.org/channel/40/Cornell_Lab_FeederWatch_Cam/, 2019. [Online; accessed 30-September-2023].

[164] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed Parallel Databases*, 33(4):479–514, 2015.

[165] B. M. Lake and R. Salakhutdinov. One-shot learning by inverting a compositional causal process. In *NIPS*, 2013.

[166] C. H. Lampert, H. Nickisch, and S. Harmeling. Learning to detect unseen object classes by between-class attribute transfer. In *CVPR*, 2009.

[167] C. H. Lampert, H. Nickisch, and S. Harmeling. Attribute-based classification for zero-shot visual object categorization. In *IEEE TPAMI*, pages 453–465, 2013.

[168] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. *Deep learning*, volume 521. Nature Publishing Group, 2015.

[169] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply supervised nets. In *AISTATS*, 2015.

[170] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large scale graph embedding system. In *MLSys*. mlsys.org, 2019.

[171] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD Conference*, pages 1517–1534. ACM, 2019.

[172] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. Training graph neural networks with 1000 layers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 6437–6449. PMLR, 2021.

[173] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *CVPR*, pages 5325–5334, 2015.

[174] Kun Li, Daisy Zhe Wang, Alin Dobra, and Christopher Dudley. UDA-GIST: An In-database Framework to Unify Data-Parallel and State-Parallel Analytics. *Proc. VLDB Endow.*, 8(5):557–568, 2015.

[175] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

[176] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[177] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[178] Side Li and Arun Kumar. Towards an optimized GROUP BY abstraction for large-scale machine learning. *Proc. VLDB Endow.*, 14(11):2327–2340, 2021.

[179] Xiao-Shuang Li, Xiang Liu, Le Lu, Xian-Sheng Hua, Ying Chi, and Kelin Xia. Multiphysical graph neural network (MP-GNN) for COVID-19 drug design. *Briefings Bioinform.*, 23(4), 2022.

[180] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. MLog: Towards Declarative In-Database Machine Learning. *Proc. VLDB Endow.*, 10(12):1933–1936, 2017.

[181] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *KDD*, pages 1754–1763. ACM, 2018.

[182] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952*, 2017.

[183] 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.

[184] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[185] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *ECCV 2014*, pages 740–755, 2014.

[186] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling GNN training on large graphs via computation-aware caching. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 401–415. ACM, 2020.

[187] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. G3: when graph neural networks meet parallel graph processing systems on gpus. *Proc. VLDB Endow.*, 13(12):2813–2816, 2020.

[188] Juncheng Liu, Kenji Kawaguchi, Bryan Hooi, Yiwei Wang, and Xiaokui Xiao. EIGNN: efficient infinite-depth graph neural networks. In *NeurIPS*, pages 18762–18773, 2021.

[189] Ziwei Liu, Ping Luo, Shi Qiu, Xiaogang Wang, and Xiaoou Tang. DeepFashion: Powering Robust Clothes Recognition and Retrieval with Rich Annotations. In *CVPR*, pages 1096–1104. IEEE Computer Society, 2016.

[190] Raoni Lourenço, Juliana Freire, and Dennis E. Shasha. Debugging Machine Learning Pipelines. *CoRR*, abs/2002.04640, 2020.

[191] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349. AUAI Press, 2010.

[192] Jiaheng Lu, Chunbin Lin, Jin Wang, and Chen Li. Synergy of Database Techniques and Machine Learning Models for String Similarity Search and Join. In *CIKM*, pages 2975–2976. ACM, 2019.

[193] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *SoCC*, 2016.

[194] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 523–534. IEEE Computer Society, 2017.

[195] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, pages 443–458. USENIX Association, 2019.

[196] MADlib. Apache MADlib: Big Data Machine Learning in SQL, Accessed 30-September-2023. https://madlib.apache.org/.

[197] MADLib. User Documentation for Apache MADlib, Accessed 30-September-2023. https://bit.ly/3epbEyS.

[198] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[199] Huizi Mao, Taeyoung Kong, and William J. Dally. Catdet: Cascaded tracked detector for efficient object detection from video. *CoRR*, abs/1810.00434, 2018.

[200] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[201] Xupeng Miao, Wentao Zhang, Yingxia Shao, Bin Cui, Lei Chen, Ce Zhang, and Jiawei Jiang. Lasagne: A multi-layer graph convolutional network framework via node-aware deep architecture (extended abstract). In *ICDE*, pages 1561–1562. IEEE, 2022.

[202] Microsoft. Azure SQL Query Processing Architecture Guide, Accessed 30-September-2023. https://docs.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver15#distributed-query-architecture.

[203] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *FAT*, pages 220–229. ACM, 2019.

[204] MLflow. MLflow, Accessed 30-September-2023. https://mlflow.org/.

[205] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *OSDI*, pages 533–549. USENIX Association, 2021.

[206] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[207] Kabir Nagrecha and Arun Kumar. Saturn: An Optimized Data System for Multi-Large-Model Deep Learning Workloads. https://adalabucsd.github.io/papers/TR_2023_Saturn.pdf, 2023. [Tech report].

[208] Supun Nakandala and Arun Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *SIGMOD Conference*, pages 1685–1700. ACM, 2020.

[209] Supun Nakandala and Arun Kumar. Nautilus: An optimized system for deep transfer learning over evolving training datasets. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 506–520. ACM, 2022.

[210] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1589–1606. Association for Computing Machinery, 2019.

[211] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.*, 0(ja), 2020.

[212] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, 2019.

[213] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, 2019.

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

[215] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *KDD*, pages 1542–1551. ACM, 2020.

[216] 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 *SOSP*, pages 1–15. ACM, 2019.

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

[218] 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.

[219] Neo4j. Neo4j, Accessed 30-September-2023. https://neo4j.com/.

[220] M. Norouzi, T. Mikolov, S. Bengio, Y. Singer, J. Shlens, A. Frome, G. S. Corrado, and J. Dean. Zero-shot learning by convex combination of semantic embeddings. In *ICLR*, 2014.

[221] State of California Department of Justice. CCPA, Accessed 30-September-2023. https://oag.ca.gov/privacy/ccpa.

[222] Shu Lih Oh, Eddie Y.K. Ng, Ru San Tan, and U. Rajendra Acharya. Automated diagnosis of arrhythmia using combination of cnn and lstm techniques with variable length heart beats. *Computers in Biology and Medicine*, 102:278 – 287, 2018.

[223] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K. H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. SINGA: A Distributed Deep Learning Platform. In *ACM Multimedia*, pages 685–688. ACM, 2015.

[224] Carlos Ordonez. Integrating K-Means Clustering with a Relational DBMS Using SQL. *IEEE Trans. Knowl. Data Eng.*, 18(2):188–201, 2006.

[225] Allen Ordookhanians, Xin Li, Supun Nakandala, and Arun Kumar. Demonstration of Krypton: Optimized CNN Inference for Occlusion-Based Deep CNN Explanations. *PVLDB*, 12(12):1894–1897, 2019.

[226] Vincent Oria, M. Tamer Özsu, Paul Iglinski, Shu Lin, and Benjamin Bin Yao. DISIMA: A Distributed and Interoperable Image Database System. In *SIGMOD Conference*, page 600. ACM, 2000.

[227] Tom O'Malley. Hyperparameter tuning with Keras Tuner, Accessed 30-September-2023. https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html?linkId=81371017.

[228] Szilard Pafka. Big RAM is Eating Big Data
- Size of Datasets Used for Analytics, Accessed 30-September-2023. https://www.kdnuggets.com/2015/11/big-ram-big-data-size-datasets.html.

[229] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proceedings of the British Machine Vision*, volume 1, pages 41.1–41.12, 2015.

[230] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. SANCUS: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.*, 15(9):1937–1950, 2022.

[231] A. Pentina and C. H. Lampert. A pac-bayesian bound for life-long learning. In *ICML*, pages II–991–II–999, 2014.

[232] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[233] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data Management Challenges in Production Machine Learning. In *SIGMOD Conference*, pages 1723–1726. ACM, 2017.

[234] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *USENIX Annual Technical Conference*, pages 631–644. USENIX Association, 2018.

[235] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. Deep Integration of Machine Learning Into Column Stores. In *EDBT*, pages 473–476. OpenProceedings.org, 2018.

[236] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[237] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8821–8831. PMLR, 18–24 Jul 2021.

[238] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.*, 11(3):269–282, 2017.

[239] Alexander J. Ratner, Braden Hancock, and Christopher Ré. The role of massively multi-task and weak supervision in software 2.0. In *CIDR*, 2019.

[240] Alexander J. Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data Programming: Creating Large Training Sets, Quickly. In *NIPS*, pages 3567–3575, 2016.

[241] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *CVPR*, pages 6517–6525, 2017.

[242] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. pages 779–788, 06 2016.

[243] Cédric Renggli, Frances Ann Hubis, Bojan Karlas, Kevin Schawinski, Wentao Wu, and Ce Zhang. Ease.ml/ci and Ease.ml/meter in Action: Towards Data Management for Statistical Generalization. *Proc. VLDB Endow.*, 12(12):1962–1965, 2019.

[244] Cédric Renggli, Bojan Karlas, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. Continuous Integration of Machine Learning Models with ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *MLSys*. mlsys.org, 2019.

[245] Alexander Renz-Wieland, Rainer Gemulla, Steffen Zeuch, and Volker Markl. Dynamic Parameter Allocation in Parameter Servers. *Proc. VLDB Endow.*, 13(11):1877–1890, 2020.

[246] Robert Ricci, Eric Eide, and CloudLabTeam. Introducing Cloudlab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[247] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488. ACM, 2013.

[248] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IJCV*, pages 211–252, 2015.

[249] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.

[250] Aécio S. R. Santos, Sonia Castelo, Cristian Felix, Jorge Piazentin Ono, Bowen Yu, Sungsoo Ray Hong, Cláudio T. Silva, Enrico Bertini, and Juliana Freire. Visus: An Interactive System for Automatic Machine Learning Model Building and Curation. In *HILDA@SIGMOD*, pages 6:1–6:7. ACM, 2019.

[251] W. J. Scheirer, L. P. Jain, and T. E. Boult. Probability models for open set recognition. In *IEEE TPAMI*, 2014.

[252] W. J. Scheirer, A. Rocha, A. Sapkota, and T. E. Boult. Towards open set recognition. In *IEEE TPAMI*, 2013.

[253] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On Challenges in Machine Learning Model Management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.

[254] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *CVPR*, pages 815–823, 2015.

[255] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*, pages 7:1–7:4. ACM, 2019.

[256] Timos K Sellis. Multiple-query Optimization. *TODS*, 1988.

[257] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *arXiv preprint arXiv:1802.05799*, 2018.

[258] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge university press, 2014.

[259] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD Conference*, pages 1171–1188. ACM, 2019.

[260] Scott Sievert, Tom Augspurger, and Matthew Rocklin. Better and faster hyperparameter optimization with Dask. 2019.

[261] Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1470–1477 vol.2, Oct 2003.

[262] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*, pages 535–546. IEEE Computer Society, 2017.

[263] Hang Su and Haoyu Chen. Experiments on Parallel Training of Deep Neural Network using Model Averaging. *CoRR*, abs/1507.01239, 2015.

[264] Jiao Sun, Mingxuan Yue, Zongyu Lin, Xiaochen Yang, Luciano Nocera, Gabriel Kahn, and Cyrus Shahabi. Crimeforecaster: Crime prediction by exploiting the geographical neighborhoods' spatiotemporal dependencies. In *ECML/PKDD (5)*, volume 12461 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2020.

[265] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *CVPR*, pages 3476–3483, 2013.

[266] Mengfan Tang, Siripen Pongpaichet, and Ramesh Jain. Research challenges in developing multimedia systems for managing emergency situations. In *ACM Multimedia*, 2016.

[267] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440. ACM, 2015.

[268] **Yuhao Zhang** and Arun Kumar. Panorama: A data system for unbounded vocabulary querying over video. *Proc. VLDB Endow.*, 13(4):477–491, 2019.

[269] Anthony Thomas and Arun Kumar. A Comparative Evaluation of Systems for Scalable Linear Algebra-Based Analytics. *PVLDB*, 11(13):2168–2182, 2018.

[270] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *OSDI*, pages 495–514. USENIX Association, 2021.

[271] S. Thrun and T. M. Mitchell. Lifelong robot learning. In *Robotics and Autonomous Systems*, volume 15, pages 25 – 46, 1995.

[272] Yuanyuan Tian. The world of graph databases from an industry perspective. *CoRR*, abs/2211.13170, 2022.

[273] Kazuyuki Tsuda, Kensaku Yamamoto, Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa. MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures. *IEEE Trans. Knowl. Data Eng.*, 3(4):444–460, 1991.

[274] Manasi Vartak and Samuel Madden. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.*, 41(4):16–25, 2018.

[275] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: A System for Machine Learning Model Management. In *HILDA@SIGMOD*, page 14. ACM, 2016.

[276] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[277] P. A. Viola and M. J. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, volume 1, pages I–I, 2001.

[278] VMware. Model Selection for Deep Neural Networks on Greenplum Database, Accessed 30-September-2023. https://bit.ly/2AaQLc2.

[279] VMware Tanzu/Pivotal. gpfdist, Accessed 30-September-2023. https://gpdb.docs.pivotal.io/510/utility_guide/admin_utilities/gpfdist.html.

[280] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks, 2022.

[281] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *ICLR*. OpenReview.net, 2022.

[282] X. Wan, Y. Luo, D. Crankshaw, A. Tumanov, and J. E Gonzalez. Idk cascades: Fast deep learning by learning not to overthink. In *UAI*, 2018.

[283] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural Deep Network Embedding. In *KDD*, pages 1225–1234. ACM, 2016.

[284] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative Deep Learning for Recommender Systems. In *KDD*, pages 1235–1244. ACM, 2015.

[285] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[286] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & Cross Network for Ad Click Predictions. In *ADKDD@KDD*, pages 12:1–12:7. ACM, 2017.

[287] Wei Wang, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *ACM Multimedia*, pages 25–34. ACM, 2015.

[288] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.*, 12(2):128–140, 2018.

[289] Wei Wang, Xiaoyan Yang, Beng Chin Ooi, Dongxiang Zhang, and Yueting Zhuang. Effective deep learning-based multi-modal retrieval. *VLDB J.*, 25(1):79–101, 2016.

[290] Pete Warden. The Machine Learning Reproducibility Crisis, Accessed 30-September-2023. https://petewarden.com/2018/03/19/the-machine-learning-reproducibility-crisis.

[291] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 84–97, 2016.

[292] Jason Weston, Frédéric Ratle, and Ronan Collobert. Deep learning via semi-supervised embedding. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 1168–1175. ACM, 2008.

[293] Gerhard J Woeginger. The Open Shop Scheduling Problem. In *STACS*, 2018.

[294] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 2019.

[295] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: A survey. *ACM Comput. Surv.*, 55(5):97:1–97:37, 2023.

[296] Tianxing Wu, Arijit Khan, Melvin Yong, Guilin Qi, and Meng Wang. Efficiently embedding dynamic knowledge graphs. *Knowl. Based Syst.*, 250:109124, 2022.

[297] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32(1):4–24, 2021.

[298] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[299] Huaze Xie, Da Li, Yuanyuan Wang, and Yukiko Kawai. Visualization method for the spreading curve of COVID-19 in universities using GNN. In *BigComp*, pages 121–128. IEEE, 2022.

[300] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.*, 12(4):446–460, 2018.

[301] Jingbo Xu, Zhanning Bai, Wenfei Fan, Longbin Lai, Xue Li, Zhao Li, Zhengping Qian, Lei Wang, Yanyan Wang, Wenyuan Yu, and Jingren Zhou. Graphscope: A one-stop large graph processing system. *Proc. VLDB Endow.*, 14(12):2703–2706, 2021.

[302] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*. OpenReview.net, 2019.

[303] Fan Yang, Ajinkya Kale, Yury Bubnov, Leon Stein, Qiaosong Wang, Hadi Kiapour, and Robinson Piramuthu. Visual search at ebay. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 2101–2110, New York, NY, USA, 2017. ACM.

[304] L. Yang, P. Luo, C. C. Loy, and X. Tang. A large-scale car dataset for fine-grained categorization and verification. In *CVPR*, pages 3973–3981, 2015.

[305] L. Yang, P. Luo, C. C. Loy, and X. Tang. A large-scale car dataset for fine-grained categorization and verification(tech report). Technical Report CNS-TR-2011-001, The Chinese University of Hong Kong, 2015.

[306] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *KDD*, pages 974–983. ACM, 2018.

[307] Atsuo Yoshitaka and Tadao Ichikawa. A Survey on Content-Based Retrieval for Multimedia Databases. *IEEE Trans. Knowl. Data Eng.*, 11(1):81–93, 1999.

[308] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. Tensor Relational Algebra for Machine Learning System Design. *CoRR*, abs/2009.00524, 2020.

[309] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. www.cidrdb.org, 2021.

[310] Ce Zhang, Jaeho Shin, Christopher Ré, Michael J. Cafarella, and Feng Niu. Extracting Databases from Dark Data with DeepDive. In *SIGMOD Conference*, pages 847–859. ACM, 2016.

[311] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, 2017. USENIX Association.

[312] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 390–404, 2017.

[313] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao. Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks. *IEEE Signal Processing Letters*, 23:1499–1503, October 2016.

[314] Quanshi Zhang and Song-Chun Zhu. Visual Interpretability for Deep Learning: A Survey. *Frontiers Inf. Technol. Electron. Eng.*, 19(1):27–39, 2018.

[315] Wentao Zhang, Yu Shen, Yang Li, Lei Chen, Zhi Yang, and Bin Cui. ALG: fast and accurate active learning framework for graph convolutional networks. In *SIGMOD Conference*, pages 2366–2374. ACM, 2021.

[316] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, KDD '18, pages 993–1001, New York, NY, USA, 2018. ACM.

[317] Yuhao Zhang and Arun Kumar. Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines. https://adalabucsd.github.io/papers/TR_2023_Lotan.pdf, 2023. [Tech report].

[318] Yuhao Zhang and Arun Kumar. Lotan: Bridging the gap between gnns and scalable graph analytics engines. *Proc. VLDB Endow.*, 16(11):2728–2741, aug 2023.

[319] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. Distributed deep learning on data systems: A comparative analysis of approaches. *Proc. VLDB Endow.*, 14(10):1769–1782, 2021.

[320] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao. PS2: Parameter Server on Spark. In *SIGMOD Conference*, pages 376–388. ACM, 2019.

[321] Zhipeng Zhang, Jiawei Jiang, Wentao Wu, Ce Zhang, Lele Yu, and Bin Cui. MLlib*: Fast Training of GLMs Using Spark MLlib. In *ICDE*, pages 1778–1789. IEEE, 2019.

[322] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. Learning on large-scale text-attributed graphs via variational inference. In *The Eleventh International Conference on Learning Representations*, 2023.

[323] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. In *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*, pages 36–44. IEEE, 2020.

[324] Zhigao Zheng, Hwa-Young Jeong, Tao Huang, and Jiangbo Shu. Kde based outlier detection on distributed data streams in multimedia network. *Multimedia Tools and Applications*, 76:18027–18045, 2016.

[325] Zhi-Hua Zhou. A brief introduction to weakly supervised learning. *National Science Review*, 5(1):44–53, 08 2017.

[326] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12):2094–2105, 2019.

[327] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized Stochastic Gradient Descent. In *NIPS*, pages 2595–2603. Curran Associates, Inc., 2010.

[328] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and Bin Xiao. Mariana: Tencent deep learning platform and its applications. 7:1772–1777, 01 2014.