An Extensible Architecture for Distributed Heterogeneous Processing

Frank Luan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-219
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-219.html

December 19, 2024

An Extensible Architecture for Distributed Heterogeneous Processing

by

Sifei Luan

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Associate Professor Matei Zaharia
Associate Professor Joseph Gonzalez
Assistant Professor Danyang Zhuo

Fall 2024

An Extensible Architecture for Distributed Heterogeneous Processing

Abstract

An Extensible Architecture for Distributed Heterogeneous Processing

by

Sifei Luan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

The exponential growth in artificial intelligence (AI) compute demands has significantly outpaced the advancement in single-node processing power. This widening gap has made distributed heterogeneous processing essential for modern AI applications. However, existing distributed data processing systems struggle to effectively handle the complexities of heterogeneous execution.

This dissertation argues for *extensibility* as the key principle in designing systems for distributed heterogeneous processing. We build two libraries on top of Ray, a distributed execution system. First, we develop the streaming batch model, which enables efficient heterogeneous execution and dynamic adaptability to varying workloads. Second, we introduce Exoshuffle, a distributed shuffle library that enables flexible control of data semantics without sacrificing performance, demonstrating that complex data operations can be implemented efficiently as application libraries rather than requiring purpose-built systems. Both libraries are integrated into the opne-source framework Ray Data, which has been adopted by thousands of companies in the industry. Finally, we validate our the effectiveness of this architecture through the CloudSort benchmark, in which Exoshuffle-CloudSort set a new world record for the most cost-effective sorting of data on a public cloud. These results demonstrate that this extensible architecture can deliver both high performance and scalability while providing the flexibility required for heterogeneous workloads. This work provides a foundation for building efficient distributed heterogeneous processing systems capable of meeting the continuously growing computational demands of AI applications.

To all my teachers.

Contents

List of Figures

List of Tables

Acknowledgments

"A teacher is one who passes on principles, imparts skills, and resolves doubts."
— *On Teachers* by Han Yu, 802 A.D.

This dissertation reflects not only my work but the extraordinary education I have received throughout my life. I dedicate this work to every teacher who has shaped my journey, for without their collective wisdom and guidance, I would not be where I am today.

First and foremost, I am grateful to my advisor, Professor Ion Stoica. I remember our first meeting at the admitted students reception in spring 2019, against the backdrop of one of the Bay Area's most spectacular sunsets from the Berkeley Hills, after a week of rain. We discussed how shuffle remained a challenge in the Spark ecosystem — a conversation that would later bloom into my first major project, Exoshuffle. To this day, I am still amazed by how Ion managed to make time for our meeting every single week, despite running a startup, serving on multiple boards, and directing the RISELab/Sky Computing Lab. He always listened to my ideas, encouraged me to pursue the ones that I was most passionate about, and steered projects in exactly the right direction to maximize their impact. "Picking the right research problem is the most important research problem" — Ion's mantra from day one has guided me since. I still smile when I remember him telling me I "have good taste in research." Perhaps the most valuable lesson I learned from Ion was his practical wisdom: "get something end-to-end working first." As he would say, if it turns out to be easy, you have created something useful; if it is hard, you have found a challenging problem whose solution will be impactful. These principles have shaped my approach to research, and will guide me throughout my professional career.

Stephanie Wang, although never officially my advisor, has become my most influential mentor in my PhD. We collaborated on every major project in this dissertation, and I have watched multiple times in awe as she transformed our rough ideas into compelling research stories. Stephanie has an almost magical ability to take the hardest part of paper writing — convincing readers why they should care about our problem — and make it look effortless. Finding someone you can work with and learn from for years is rare, and I count myself very lucky to have found that in Stephanie. When I struggled, she knew exactly when to encourage me and when to give me that extra push I needed. To Stephanie's future students: you are in for an amazing journey.

I am honored that Joseph Gonzalez, Matei Zaharia, and Danyang Zhuo agreed to serve on my dissertation committee. Joey has this wonderful habit of turning every conversation into new connections — I always left our talks with a list of brilliant people I needed to meet. I was humbled to learn that my desk at the lab used to be Matei's; I hope I have done it justice. Many of my work involved Spark as a comparison baseline, and presenting those results to its creator was nerve-wracking at first, but Matei's feedback always helped us push our work further. Danyang was among the first to welcome me into the lab as a

postdoc in winter 2020, and invited me to dinner to make sure I was not alone on my first Chinese New Year's Eve at Berkeley. I am grateful for all of their feedback and support in shaping this dissertation.

The work in this dissertation represents a collective achievement made possible by many brilliant collaborators: Ziming Mao, Ron Yifeng Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Derek Luan, Amog Kamsetty, Eric Liang, Romil Bhardwaj, Wei-Lin Chiang, Woosuk Kwon, Michael Luo, Gautam Mittal, Zhanghao Wu, Zongheng Yang, and Siyuan Zhuang. Many of you have become dear friends, and watching you succeed in your own journeys brings me incredible joy. I also want to thank everyone at Anyscale — your contributions to Ray Data and the broader Ray project have helped our work reach farther than I ever imagined.

I like to tell people that my PhD experience was filled with joy and happiness, rather than isolation and stress. I owe that happiness to the countless friends I made at the Sky Computing Lab and across Berkeley. While I cannot possibly capture all our memories in these pages, I hope you know how much our time together has meant to me.

Before Berkeley, I worked at Facebook from 2017 to 2019, where I was very fortunate to join a cutting-edge research team straight out of college. I would like to think of my first manager, Dr. Satish Chandra, as my first real research advisor. He mentored me and a couple of other college graduates into prolific researchers that published numerous pioneering papers in applying machine learning to software engineering practices [81, 52, 12, 20]. I am still amazed by how Dr. Erik Meijer, the director of the research group, managed to create an oasis of pure research in an otherwise fast-paced and competitive Silicon Valley company. I am also grateful to Professor Koushik Sen, who not only guided my early research steps but also introduced me to RISELab at Berkeley, opening the door to this incredible journey.

As I reflect on those who guided me onto this path, I find myself fortuante to have met some of the most caring and supportive teachers during my formative years. They saw something in me that I couldn not yet see in myself, and have been my most enthusiastic champions ever since. Ms. Song Lirong, my headmaster in third and fourth grade in elementary school, saw the potential in me and sent me to one of the best math schools in Beijing. More than twenty years later, she still follows my journey with the same warmth and enthusiasm. When I recently discovered that she had written about me in her book on child education [86], I was deeply moved by how vividly she remembered details of my childhood that even I had forgotten. My middle school headmaster, Ms. Wu Ling, taught me what it means to live with love, respect, and integrity. I have carried these principles with me ever since. In high school, Ms. Li Jing, our vice principal, opened windows to the world for me. She went to great lengths to create opportunities for me to engage with international scholars, filmmakers, and entrepreneurs, and encouraged me to apply to the best universities in the world. Mrs. Sally Levy gave me the confidence and support I needed to apply to US colleges. At the University of Chicago, Dr. Eugenia Cheng taught my first calculus class. She taught me to see mathematics not just as a tool, but as a pure and beautiful language

for understanding the world. Her passion for mathematical rigor and elegant proofs planted the seeds for me to become a researcher. Years later, I discovered she had included a thank-you letter I wrote to her in her book *How to Bake Pi* [23]. It felt like coming full circle — my words living alongside the teachings that had so profoundly shaped my path. These extraordinary teachers brought out the best in me, and their influence continues to shape who I am today.

I was fortunate to have met my partner Yifei Li in 2023, who transformed my life ever since. We have been through ups and downs together, and I am grateful for her believing in me, supporting me, and showing me the beauty of life that I would have never seen by myself.

Above all, I owe everything to my parents: my mother Hongyu Guo, and my father Jun Luan. When I was born, they carried three hopes for me: that I would grow up healthy, that I could develop a special talent, and that I would not do harm to society. They did not just support my dreams — they gave me the wings to chase them. Throughout my PhD journey, they have been my most steadfast supporters. Every achievement of mine is a reflection of their unwavering love and support.

To my parents, my family, my mentors, and all my teachers: I dedicate this work to you.

Chapter 1

Introduction

The fundamental challenge in computer system design has always been reconciling the growing demands of applications with the constraints of available hardware. As hardware capabilities evolve and application requirements become more sophisticated, system designs must adapt to address new computing paradigms. In this dissertation, we examine system design for one such emerging paradigm, driven by the latest growth in artificial intelligence (AI) applications: distributed heterogeneous processing.

Growth in AI Compute Demands The past decade has seen unprecedented growth in AI applications. The deep learning revolution, beginning in 2012, established neural networks as the dominant paradigm for tasks ranging from image recognition to natural language processing. The state-of-the-art models of this era, such as AlexNet and Word2Vec, had about 10^8 parameters, and the training compute required around 10^{17} FLOPs. A decade later, the state-of-the-art large language models, such as Llama 3, have about 10^{11} parameters, and the training compute required is in the order of 10^{25} FLOPs. This amounts to an exponential growth of approximately $4\times$ year-over-year, or $10^8\times$ over the past 12 years.

Modern large language models demonstrate capabilities far beyond their predecessors, including few-shot learning, multi-modal understanding, and complex reasoning. However, these advances in model capability have come at the cost of unprecedented computational requirements.

Growth in Hardware Performance On the hardware side, however, the growth in single-node processing power has been remarkably slower. Single-thread CPU performance has grown by merely $1.1\times$ year-over-year since the early 2000s, constrained by the end of Dennard scaling and the slowdown of Moore's Law.

The industry has since shifted to specialized hardware to meet the growing compute demand of AI applications. Modern GPUs, such as NVIDIA's H100, provide massive parallelism through thousands of CUDA cores and high memory bandwidth, both especially

beneficial for deep learning training. Other special-purpose accelerators, like Google's Tensor Processing Unit (TPU) and Amazon's Trainium and Inferentia chips, offer specialized architectures for neural network computations. Despite these accelerators providing 2–3 orders of magnitude higher performance compared to CPUs for AI workloads, the growth rate of their performance is still outpaced by the faster-growing compute demands of AI applications. For example, GPU performance has been growing at around 1.5× year-over-year (following Huang's law), which is still slower than the 4× annual growth rate of AI compute requirements.

The widening gap between the growth rate of AI compute demands and the growth rate of hardware performance has made two trends apparent:

1. **More applications will require distributed processing.** As individual hardware nodes cannot keep pace with compute requirements, applications increasingly require distributed execution across multiple nodes.
2. **More applications will require heterogeneous computing.** Modern AI applications handle diverse data modalities and complex processing pipelines, requiring different hardware accelerators for different stages—from CPU-based data preprocessing to GPU-accelerated model training and inference.

Both these trends present unique challenges that existing distributed data processing systems were not designed to address. Next, we discuss how these systems struggle with the complexities of heterogeneous execution and the dynamic nature of modern data and AI applications.

1.1 Challenges in Distributed Heterogeneous Processing

Let us consider a distributed heterogeneous data processing program expressed as a directed acyclic graph (DAG) of operators. In this graph, different operators may require different hardware resources—CPUs, GPUs, or other accelerators. Data flows through this DAG in batches (or partitions). Optimizing throughput in such a system requires maintaining high utilization across heterogeneous compute resources while addressing several key challenges:

- The system must manage efficient operator execution across heterogeneous resources.
- The system must coordinate data movement across memory hierarchies, network boundaries, and storage systems.

- The system must provide robust failure recovery mechanisms for partial failures, including process and node failures.

Next, we examine two major categories of existing distributed data processing systems—batch processing and stream processing—and discuss the distinct challenges they face when handling heterogeneous workloads.

Batch processing systems In the batch processing model, such as MapReduce [26] and Spark [105], programs execute in stages, with data-parallel operators processing data in partitions. Data repartitioning (shuffle) occurs between stages. This execution model presents two significant challenges for heterogeneous processing.

One issue is the high cost of materializing intermediate data. In heterogeneous processing, different stages often require different hardware accelerators, leading to varying resource requirements. In the batch processing model, repartitioning, i.e., changing data parallelism, can only happen at stage boundaries. This means that data must be materialized to disk between stages. This becomes particularly problematic in multi-modal data processing, such as video processing, where intermediate data can be 1000× larger than the input data. This explosion in data size makes the I/O cost of materialization prohibitively expensive.

Another problem is rigid output semantics. Batch processing systems implement a stage-by-stage execution model, in which no output is available until the entire dataset is processed. This makes them unsuitable as data loaders for ML training, which requires the output to be produced in a continuous stream to keep the accelerators busy.

Stream processing systems Stream processing systems, such as Flink [21], are optimized for low-latency, stateful processing of continuous data streams. While these systems excel at real-time data processing, they face significant challenges when applied to throughput-oriented heterogeneous data processing:

The first problem is resource allocation. Unlike batch processing systems, which execute tasks in a pool of executors, stream processing systems require operators to be statically allocated resources at the beginning of the job. This static resource allocation becomes problematic with ML workloads, particularly with multi-modal data, where workload variability is common. The difficulty in reconfiguring operator parallelisms without system restart makes it challenging to adapt to changing resource requirements.

Secondly, failure recovery is expensive. Stream processing systems usually implement checkpoint-based recovery, requiring periodic state checkpoints and system-wide restarts from the last checkpoint upon failure. This approach is suitable for stateful actors. However, it makes fault recovery from individual stateless task failures, e.g. due to out-of-memory errors, unnecessarily heavy-handed.

To summarize, existing distributed data processing systems are built as monolithic systems and optimized for specific data processing paradigms. The design decisions baked into these systems make them difficult to extend for heterogeneous processing. In other words, they are not *extensible* to evolving hardware and application requirements.

1.2 An Extensible Architecture

This dissertation addresses the question of how to built systems that effectively support distributed heterogeneous processing. We argue that *extensibility* is the key design principle, manifesting in the following three aspects:

- **Flexibility:** The system must be flexible to support diverse execution patterns to address the needs of heterogeneous execution, including staged batch processing, stream processing, and a hybrid of these approaches.
- **Dynamic Adaptability:** The system must be able to dynamically adapt to changing execution patterns to accommodate both workload variability and resource elasticity.
- **Interoperability:** Applications built on top this system should be inter-operable, i.e. they must be able to efficiently communicate and share data.

In addition to these extensibility requirements, the traditional requirements for distributed data processing systems remain essential: the system must provide high performance, scale effectively to large clusters, and provide robust fault tolerance facilities.

To address these requirements, we implement our distributed heterogeneous processing solutions as application libraries on top of Ray, a distributed execution system designed with extensibility as its core principle. Ray's architecture provides several key capabilities that make it an ideal foundation for our work:

- A flexible task execution model with simple yet powerful APIs for distributed computation. Through these APIs, developers can schedule both stateless functions (as *tasks*) and stateful classes (as *actors*), while precisely controlling *when* tasks execute using synchronization primitives and *where* they run by specifying resource requirements.
- An efficient *distributed futures* system that transparently manages data objects across the cluster. This abstraction allows programs to pass object references between tasks, while the system handles the complexity of memory management and cross-node data transfer.
- Robust fault tolerance through lineage-based recovery mechanisms. The system's distributed memory object store architecture decouples task execution failures from system-level failures, improving reliability when processing large-scale workloads.

- Deep integration with the Python ecosystem, including native support for popular data and ML frameworks such as NumPy, SciPy, PyTorch, and JAX, enabling interoperability with existing ML applications.

For a comprehensive view of Ray's architecture and implementation details, we refer readers to Stephanie Wang's dissertation *Towards a Distributed OS for Data-Intensive Cloud Applications* [97].

Dissertation Contributions In the rest of this dissertation, we present two application libraries built for distributed heterogeneous processing, followed by a case study showcasing the libraries' performance and scalability.

In Chapter 2, we introduce the *streaming batch* model, a novel execution model that enables efficient heterogeneous processing. It features an adaptive scheduler that enables efficient heterogeneous execution, even when the working set is much larger than available memory. Compared to batch processing, it enables full utilization of heterogeneous resources while avoiding materializing intermediate data; compared to stream processing, it is much more adaptive to varying workloads by allowing fast reconfiguration of operator parallelisms. We implement this model in Ray Data, an open-source framework for ML data pre-processing. The Ray Data framework is widely adopted in ML training and batch inference workloads, including the pre-training of the Stable Diffusion model on a billion-image dataset.

The streaming batch model is suitable for map-style data pipelines. In Chapter 3, we build *Exoshuffle*, a library for complex data operations that involve distributed shuffle. Previously, purposely-built shuffle systems were required to provide efficient shuffle operations to data processing frameworks. In *Exoshuffle*, we demonstrate that by decoupling the shuffle control plane from the data plane, shuffle can be implemented in 10× less code without sacraficing performance. Furthermore, running shuffle as an application library enables new applications such as ML training to leverage scalable shuffle, providing greater flexibility and interoperability than previous shuffle solutions.

Finally, in Chapter 4, we demonstrate the performance and scalability of the *Exoshuffle* architecture by running the CloudSort benchmark. *Exoshuffle-CloudSort* set the new world record for the most cost-effective way to sort 100 TB of data on the public cloud, costing less than $1 per TB.

Together, these contributions advance the state of the art in distributed heterogeneous processing, providing a foundation for building efficient and scalable systems that can meet the growing demands of modern AI applications.

Chapter 2

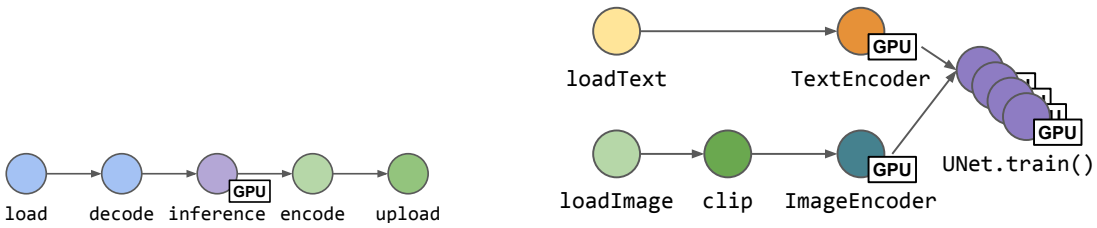
The Streaming Batch Model for Efficient Heterogeneous Execution

As discussed Chapter 1, distributed data processing systems based on the batch or stream processing models are designed primarily for homogeneous execution environments. When running applications that demand heterogeneous execution resources, they often result in under-utilization of resources, especially when memory is limited.

In this chapter, we introduce the *streaming batch* execution model, a hybrid of the batch and stream processing models that enables efficient heterogeneous execution. The key idea is to execute one *partition* at a time to allow dynamic allocation of resources. This enables pipelining across heterogeneous resources, similar to the stream processing model, but offers the resource multiplexing, elasticity, and fault tolerance properties of the batch processing model. We present Ray Data, an implementation of the streaming batch model that can adaptively repartition and re-allocate resources based on actual run-time usage. We show that Ray Data improves throughput on heterogeneous batch inference pipelines by 3–8× compared to traditional batch and stream processing systems. On heterogeneous training and inference pipelines such as Stable Diffusion, Ray Data matches the single-node throughput of ML-specific data loaders while additionally leveraging distributed and heterogeneous clusters to further improve training throughput by 31%.

2.1 Introduction

Data processing is critical to machine learning applications. While model training and inference are both GPU-intensive, they also require significant I/O and CPU to load and preprocess datasets. CPU-based preprocessing is often the bottleneck in both training [62] and batch inference [45]. Meanwhile, as ML models evolve, the data processing functionality required has also become more diverse, spanning many modalities, transformations, and resource requirements [41, 92, 66].



(a) Batch inference.

(b) Training Stable Diffusion.

Figure 2.1: Heterogeneous applications in batch inference and training, represented as logical dataflow graphs (nodes are operators). (a) represents a typical pipeline for video or image generation. (b) represents a pipeline for training stable diffusion model. In (b), `UNet` model is replicated for data-parallel training.

There are numerous frameworks designed to scale out CPU-based data processing, typically based on either the batch [9, 105, 26] or stream [60, 21, 46, 2] processing models. These systems allow users to express a dataflow of logical *operators* (Figure 2.1), while the system automatically handles data distribution, task scheduling, and fault tolerance. Data processing in ML pipelines often consists of pure map transforms and thus can easily be expressed with this API [62]. However, these systems were designed for homogeneous CPU-only clusters. Heterogeneous ML pipelines have two key properties that limit performance and scalability when executed on CPU-centric data processing frameworks.

First, different operators require different degrees of physical parallelism, i.e. the number of concurrently executing instances. Thus, a key system requirement is to *decouple the physical parallelism of each operator*. For example, one may use many CPU threads to download an input dataset from cloud storage to match the throughput of a single GPU. This requires buffering intermediate data in memory until the downstream operator is ready to consume them. As data modalities such as text, images, and video proliferate [66, 88, 36, 80], the intermediate data size can be significant and unpredictable. Meanwhile, memory is more challenging to multiplex, as oversubscription is costly. Maintaining throughput while staying under memory limit is critical.

Second, some resources are more valuable than others. Thus, it is critical to *isolate failure domains to the failed resource*. For example, GPUs are more expensive than CPUs, but CPU failures are more likely because they are deployed in higher numbers and more often on spot instances [102]. Ideally, a CPU failure should have little impact on GPU execution. This is challenging to achieve simultaneously with exactly-once record processing and minimal runtime overheads. Even with stateless transforms, the system must track which records have been processed for each operator to avoid duplicating or dropping records during failover.

Thus, a distributed data processing system for ML pipelines must: (1) dynamically adjust each operator's parallelism according to actual compute and memory usage, while (2)

minimizing run-time and recovery overheads from failures and dynamic re-scaling. Current batch and stream processing systems can achieve one but not both. Batch processing systems execute operators in *synchronous* stages of stateless and immutable tasks, replaying them upon failure. This method, known as lineage reconstruction, offers high fault tolerance and elasticity but limited support for dynamic parallelism [26, 105]. Stream processing systems execute operators with *asynchronous* stateful executors that can decide locally when to materialize records. However, this decentralized approach requires a recovery method that imposes either high run-time overhead from logging [2, 60] or high recovery overhead from global checkpointing and rollback [21, 60].

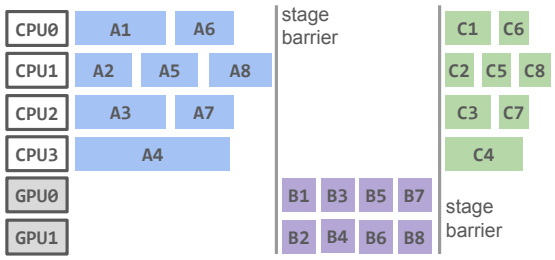
To address these challenges, we present the *streaming batch* model, a hybrid model for efficient and fault-tolerant heterogeneous execution. Similar to the batch processing model, a centralized scheduler partitions data across a stage of stateless tasks. Tasks can run on any executor and are recovered via lineage reconstruction, enabling elasticity and fault tolerance while avoiding unnecessarily expensive logging. However, similar to the stream processing model, stages are executed *asynchronously*, allowing data to be streamed to the next operator and *dynamically repartitioned*.

We present Ray Data, a distributed streaming batch system for heterogeneous workloads such as batch inference and ML training. Ray Data implements a logical dataflow API with an ahead-of-time execution planner and a run-time scheduler. The execution planner decides the initial number of partitions per operator. During execution, operators decide when to materialize a task's intermediate partitions, allowing dynamic repartitioning based on local memory usage. Meanwhile, the centralized scheduler maintains a global view of running tasks and materialized partitions and can accordingly adjust the physical parallelism of each operator, i.e. its memory and compute allocation, while enforcing overall memory limits.

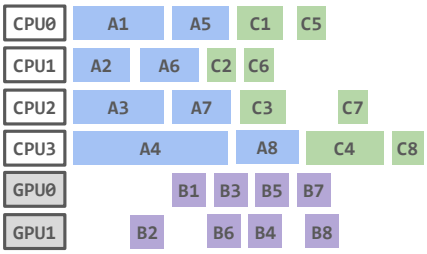
Ray Data uses Ray [59] as a distributed task backend. We extend Ray's lineage-based recovery with a key feature needed for streaming batch execution: dynamic repartitioning.

We evaluate Ray Data on batch inference and ML training workloads that span diverse resource requirements (CPUs vs. GPUs), storage (local disk vs. cloud), and modalities (image vs. video). Ray Data outperforms batch and stream processing systems such as Spark and Flink with 3–8× better throughput. Ray Data is also able to match the throughput of single-node ML-specific data loaders such as tf.data and PyTorch DataLoader while additionally leveraging distributed and heterogeneous clusters. On a Stable Diffusion training benchmark, Ray Data can improve training time by 31% by leveraging a pool of 72 heterogeneous GPUs. In summary, we contribute:

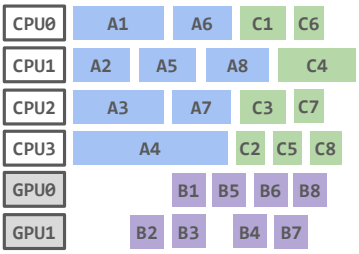
- The streaming batch model, an efficient and fault-tolerant execution model for distributed heterogeneous processing.
- An online and heterogeneity-aware scheduling policy for streaming batch systems that can enforce total memory limits and maximize total compute utilization.



(a) Batch processing model.



(b) Stream processing model.



(c) Streaming batch model (Ray Data).

Figure 2.2: Execution models for Figure 2.1a, after fusing homogeneous operators into a 3-stage pipeline: `A`→`B`→`C`. Numbers are partition indexes, e.g., `B1` depends on `A1`. **(a)** Batch processing executes one stage at a time, materializing all intermediate outputs. **(b)** Stream processing pipelines across heterogeneous resources but have fixed parallelisms: Each executor executes a fixed set of operators for a fixed key range. **(c)** The streaming batch model can reassign resources for each partition, improving resource utilization and maintaining memory efficiency.

- Ray Data, a fault-tolerant, memory-efficient, and autotuning implementation of the streaming batch model.

2.2 Background

We overview key aspects and limitations of the batch and stream processing models, including ML data loaders, by analyzing how effectively each system can:

- Manage memory for intermediate data between operators.
- Maximize utilization of heterogeneous compute resources.
- Minimize overheads for cluster failures and re-scaling.

2.2.1 Applications

We target ML training and inference pipelines that require data pre- and post-processing using CPUs, GPUs, or both. Our goal is maximizing throughput, while providing sub-second latency when used as data loaders for ML training [62]. Similar to current ML dataloaders [62, 73], we primarily target map-style per-row transforms. Operations that require all-to-all shuffle exchanges, such as sort and group-by, are also supported in the system but are already discussed in [51].

Figure 2.1a shows a typical batch inference pipeline, which uses CPUs to load and filter a dataset from cloud storage (e.g., HDFS or S3), GPUs to produce predictions, then CPUs to upload the results. A key characteristic is that the different operators may require different degrees of parallelism. For example, `load` may require many CPU threads to download an input dataset from cloud storage, while `filter` may only require one per core. Also, CPUs typically vastly outnumber GPUs in a cluster, so `predict` runs at much lower parallelism.

Figure 2.1b shows a typical distributed training pipeline for the Stable Diffusion model [80]. The pipeline uses CPUs to load and preprocess image-text pairs, GPUs to produce encodings with a pre-trained `Encoder` model, then GPUs to train a `UNet` model. Even GPU operators can leverage heterogeneity: colocating `Encoder` and `UNet` on the same GPUs can lead to lower training throughput, as it takes valuable resources from the `UNet`. If the operators' physical resources are decoupled, then `Encoder` can be run on lower-end GPUs. This reduces overall cost by increasing utilization on the `UNet` GPUs.

2.2.2 Batch Processing Model

Batch processing systems are designed to allow a task to run and therefore recover on any executor. The system transforms the user-provided logical DAG (Figure 2.1a) into a physical DAG of stages of data-parallel tasks (Figure 2.2a). Tasks are stateless and materialize their input and output partition(s). This is key to lineage-based recovery, which logs only the logical DAG and re-executes tasks to recover lost partitions. Elastic scaling is supported by simply adding or removing executors. Examples include MapReduce [26], Apache Hadoop [9], Apache Spark [105], Spark Streaming [106], and Apache Flink in `BATCH` execution mode [21].

However, to make this recovery method practical, the system imposes two significant restrictions on execution. First, each stage must fully execute and materialize its outputs *before* executing the next stage. This simplifies scheduling and recovery, as a (re)scheduled task never idles waiting for its inputs. To reduce overheads from materialization, consecutive map operators are often fused into one stage [105]. This is effective when operators require the same resources. For example, suppose the `load` and `filter` operators in Figure 2.1a are fused. If each uses 1 CPU, then this has no impact on compute utilization, but greatly reduces the memory footprint, as data can be loaded and filtered one batch at a time.

Unfortunately, when different stages require different resources, operator fusion can cause significant underutilization. For example, with 4 CPUs and 2 GPUs, if Figure 2.1a fused *all* operators, then we could execute at most two tasks at a time. This would underutilize the CPUs while leaving the GPUs idle during CPU execution. Disabling fusion of heterogeneous operators, as in Figure 2.2a, is also imperfect, as it requires materializing each stage's outputs. This prevents pipelining between CPUs and GPUs, while producing high memory pressure and likely disk spilling.

Second, the data partitioning must be determined *before* execution so that it can be recorded in the lineage. This prevents the system from using run-time information such as the in-memory size of intermediate data rows when deciding the partitioning strategy. Thus, even if pipelined stage execution were supported, there can still be high memory pressure from individual intermediate partitions that are too large. For example, suppose that Figure 2.1a were run on a dataset of 100 videos, each 10MB on disk but 20GB decoded [15]. The system may choose 100 partitions to ensure good load-balancing for `load`, but each task would require 20GB memory! While some control is exposed to the user, typically one can only specify a target number [105], again before execution. Thus, it is left to the user to predict each operator's memory usage, then manually configure the number of partitions.

2.2.3 Stream Processing Model

Stream processing architectures optimize for online scenarios and asynchronous execution. Examples include Naiad [60], Apache Flink [21], Spark Continuous Processing [93], MillWheel [2], and Apache Kafka [46]. Typically, each logical operator is assigned a parallelism, by the user or the system, which determines how many *physical operator* instances to create. For example, the parallelism of `A-B-C` in Figure 2.2b is 4-2-4. Consecutive logical operators that have the same resource requirements and parallelism are fused to avoid materialization. Physical operators execute asynchronously on their input stream(s) and exchange record batches directly, without involving a centralized scheduler.

Each physical operator processes a pre-determined stream partition. For example, in Figure 2.2b, `A8` is pre-assigned to CPU3. In contrast, Figure 2.2a executes `A8` earlier, on *any* free CPU. However, unlike batch processing systems, physical operators are stateful and can choose at run time how many records to process at a time. For example, in Figure 2.2b, the system assigns CPU3 1/4 of the total key range for operators `A` and `C`, but the executor chooses when to materialize `A4` vs. `A8` and `C4` vs. `C8`. Typically, the executor will accumulate records up to a time or memory limit, then materialize and send the batch to a downstream executor. If the downstream executor is overloaded, backpressure is applied to limit memory.

Physical operators are tied to an executor and assigned one global partition, so reconfiguration is important for load-balancing. Reconfiguration efficiency is a common challenge [94], and especially so for heterogeneous pipelines. First, for heterogeneous operators, fusion is often impractical. For example, Figure 2.2b does not fuse `A` and `C` because `B` requires a GPU.

Determining the optimal parallelism for `A` and `C` can be challenging without run-time information. Second, heterogeneous clusters can greatly reduce cost, but efficient elastic scaling and failure handling are critical.

There is a common tradeoff among stream processing systems between run-time overheads vs. reconfigurability and recovery overheads. Many systems use asynchronous *global checkpointing* [60, 21]. This typically imposes low execution overheads, but *any* failure causes a global rollback to the last checkpoint. Also, global checkpoints must coordinate each process's local checkpoints, so static membership is often assumed for simplicity [30]. Thus, reconfiguration typically introduces long pauses, as it requires taking then restarting from a global checkpoint with the new configuration [21].

Other systems use *logging* [2, 46]. This allows for fast recovery via log replay and fast repartitioning of physical operators. However, intermediate records must be durably logged before releasing to the downstream operator, adding higher run-time overheads. This is acceptable for online systems that interact frequently with the external world, but ML pipelines are typically offline systems where most intermediate values are safe to rollback and thus do not need durability.

ML data loaders. ML-specific data loaders are single-node systems that maximize I/O and CPU bandwidth to improve local GPU utilization. Examples include tf.data [62] and PyTorch DataLoader [73]. These systems can be viewed as a special case of stream processing: they launch a fixed pool of worker threads or processes at run time, which continuously load, preprocess, and feed data to an end GPU consumer. The pool must be tuned to match the GPU's throughput without running out of memory. tf.data automatically adjusts operator parallelism at run time to maximize GPU utilization [62].

However, in general these data loaders are narrow in scope: (1) they do not support distributed execution, and (2) GPUs are always assumed to be sinks. (1) prevents load-balancing and heterogeneous clusters. Both (1) and (2) make it impossible to leverage heterogeneous GPUs in the Stable Diffusion example in Figure 2.1b, as well as batch inference pipelines that alternate CPU and GPU operators. Adapting data loaders to support these applications would require effort equivalent to re-building a distributed data processing framework.

2.3 Overview: The Streaming Batch Model

The streaming batch model (Figure 2.2c) uses task-based execution with a centralized scheduler. Tasks can run on any executor, similar to batch processing (Figure 2.2a). However, tasks across different stages are pipelined and dynamically repartitioned (Figure 2.3b), similar to stream processing systems (Figure 2.2b). Table 2.1 shows a full feature comparison.

	Batch [26, 9, 105]	Stream [60, 21, 2, 46]	PyTorch DL [73]	tf.data [62]	Streaming batch (Ray Data)
Automatic partitioning	✓	✓	×	✓	✓
Dynamic repartitioning	×	✓	×	×	✓: streaming repartition
Min. pipeline granularity	Stage	Partition	Partition	Partition	Partition
Dynamic parallelism	✓	✓/×	×	✓	✓
Distributed execution	✓	✓	×	×	✓
Fault tolerance method	Lineage	Logging/Checkpointing	None	Checkpointing	Lineage
Min. rollback granularity	Partition	Record/Epoch	Epoch	Epoch	Partition

Table 2.1: Features for heterogeneous and distributed processing. Dynamic repartitioning is important for memory efficiency. Finer pipeline granularity improves utilization for heterogeneous resources. Dynamic parallelism is important for adaptivity. Stream processing systems use logging for dynamic parallelism with zero downtime and record-level rollback, or checkpointing, which requires a checkpoint to reconfigure and global rollback on failure (§ 2.2.3).

There are two primary challenges: (1) extending lineage reconstruction to support streaming batch execution, and (2) building a centralized scheduler that adaptively manages all running tasks and intermediate partitions.

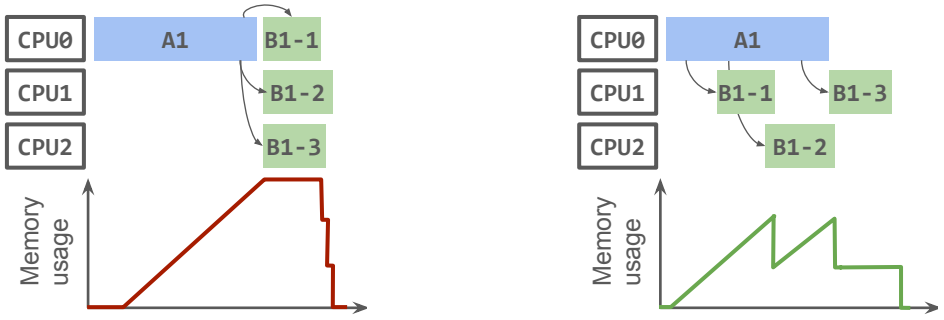
Challenge 1: Fault-tolerant, memory-efficient partitioning. Batch processing tasks are often memory-inefficient when operators require different parallelisms. For example, Figure 2.3a shows a two-stage pipeline where parallelism=1 and 3 for stages `A` and `B`, respectively. Repartitioning `A1` requires a stage barrier, causing high memory usage.

Our goal is to achieve the memory efficiency of stream processing but maintain the low run-time and recovery overheads of lineage-based recovery. In particular, we want to allow the user to set a *single* target partition *size*, instead of requiring the user to estimate the *number* of partitions per operator. This is challenging because then the number of partitions is unknown until run time. Meanwhile, lineage-based systems typically require logging the operations *before* execution.

To address this, we propose a *streaming repartition* (§ 2.4.2.1): a task may dynamically generate multiple output partitions, based on its real-time memory consumption. For example, in Figure 2.3b, if `A1` experiences memory pressure, it can output a partition early. As we also support pipelined stage execution, this reduces peak memory usage: `B` tasks can begin and release their input partitions while `A1` is executing.

For recovery, we extend Ray's lineage-based recovery [99] (§ 2.4.2.2). Ray includes a distributed object store, which we use for intermediate partitions, and task-parallel execution. Like other lineage-based systems, Ray requires immutable task definitions and does not allow task outputs to be read before task completion. To support streaming repartition, we add support in Ray for tasks with dynamic and streaming outputs.

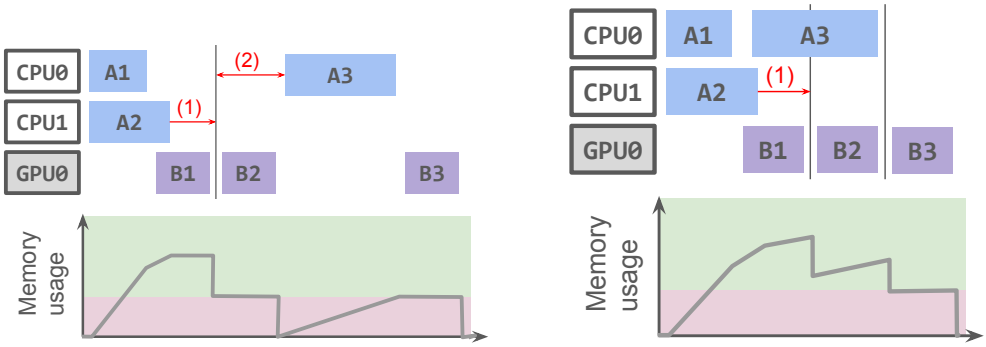
Challenge 2: Dynamic reconfiguration. One advantage of the streaming batch model is that it uses a centralized scheduler to schedule all tasks, giving the scheduler a global



(a) Repartitioning in batch processing.

(b) Streaming repartition.

Figure 2.3: Streaming repartition (b) allows executors to: (1) locally decide when to output partitions, and (2) pipeline execution with the next operator. Overall, this reduces peak memory usage compared to repartitioning in batch processing systems (a).



(a) Pessimistic scheduling.

(b) Optimistic scheduling.

Figure 2.4: Scheduling under memory pressure. Green represents CPU executors' local memory capacity (1 partition per CPU, 2 total). Pink represents the system's shared memory capacity for intermediate data (1 partition total). (1): If shared memory is full, executors must stall and buffer outputs locally until space is made, either via spilling to disk or executing downstream tasks. (b) is faster because it schedules `A3` as soon as possible but doesn't stall CPU0.

view of all resources. This enables enforcing a hard limit on memory used by intermediate data partitions. However, this must be done without unnecessarily stalling compute tasks. Memory is shared among all operators, and oversubscribing memory can cause extreme slowdowns from stalls or spilling to disk. When memory is limited, the choice of which task to execute and when is not obvious.

For example, consider Figure 2.4a. Each CPU executor has local memory capacity (green) for one partition, and the system has shared memory capacity (pink) for one partition. In phase (1), `A2` must stall and buffer its outputs locally until `B1` completes. A conservative

Method	Description
`read`	Read items from files.
`map`	Transform each item.
`map_batches`	Transform a batch of items. Useful for controlling GPU batch size.
`flat_map`	Transform each item and flatten the results.
`filter`	Return items that match a predicate.
`limit`	Truncate to the first N items.
`write`	Write items to files.
`iter`	Return an iterator of items.
`iter_split`	Split into N iterators.
`materialize`	Materialize all items.

Table 2.2: A subset of the Ray Data `Dataset` API. The bottom four are consumption APIs that trigger execution, while the others are lazy.

scheduler additionally waits for phase (2) before scheduling `A3`, to avoid stalling CPU0. Scheduling `A3` optimistically so that it finishes simultaneously with `B2`, reduces overall run time (Figure 2.4b). Applying such optimizations requires dynamic profiling and reconfiguration.

To address this challenge, we introduce an adaptive scheduler in Section 2.4.3. The key idea is to fairly allocate shared compute resources between operators, while estimating future memory availability from run-time information to enable optimistic scheduling.

2.3.1 The Dataset API

A `Dataset` represents an application pipeline. `Datasets` are lazily created, by reading files or applying transforms to an existing `Dataset` (Table 2.2). A `Dataset` is materialized through a `write` operation, e.g., to cloud storage, or by iterating over the items in memory.

A key part of the API is the ability to express *resource requirements*. Resource requirements are a map from resource name to float value and may be passed as an option to the transforms. By default, each transform requires 1 CPU. Resource names can be `CPU`, `GPU`, or a custom resource label.

Most of the map-style transforms take a stateless and pure user-defined function (UDF) as an argument. For operations that require significant initialization time, such as a model loaded into GPU memory, we also support stateful UDFs that can be instantiated once and called multiple times on different items. We assume that all UDFs are pure, to enable lineage-based recovery.

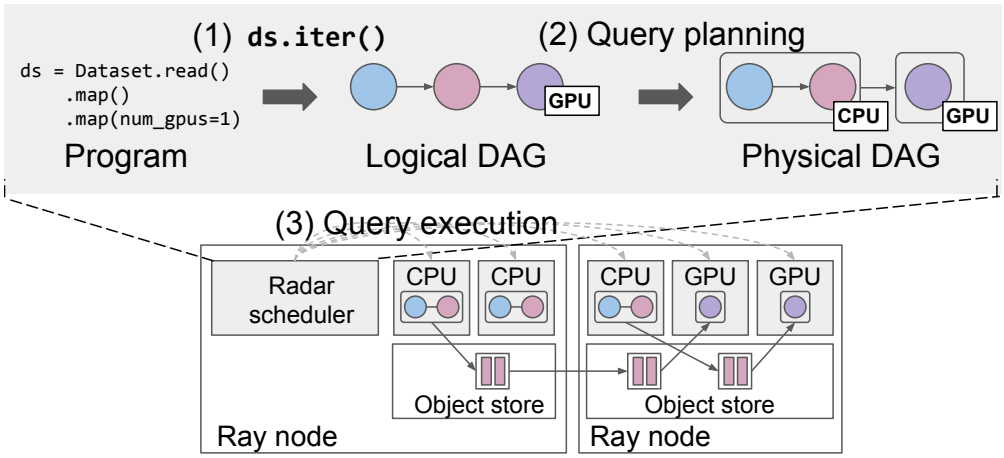


Figure 2.5: Ray Data architecture overview. Ray Data executes as a Ray library. The Ray Data scheduler executes as a Ray driver, dispatching tasks to Ray workers (dashed arrows).

2.3.2 Executing a Ray Data Program

After creating a `Dataset`, the user triggers execution by calling one of the consumption APIs, as seen in (1) in Figure 2.5. The `Dataset` is represented as a DAG of logical operators, as in Figure 2.1. The system's query planner then compiles this logical DAG into a DAG of physical operators ((2) in Figure 2.5). The query planner applies operator fusion and decides the number of partitions to use for the first operator (§ 2.4.1). Each physical operator defines a transform to apply to each row and metadata such as the resources required. Physical operators are decoupled from executors, and their parallelism may not be determined until run time.

Ray Data uses Ray as a task backend, storing intermediate data partitions in Ray's distributed object store. During execution, the Ray Data scheduler dispatches each physical operator as one or more Ray tasks ((3) in Figure 2.5). Stateless UDFs are executed as Ray *tasks*, which are Python processes that can run anywhere in the cluster. Stateful UDFs are executed as tasks running on Ray *actors*, which are long-running Python processes that preserve state. Actors acquire resources for their lifetime. Thus, if multiple stateful UDFs require the same resource, multiplexing is possible by sharing an actor pool between UDFs.

For each physical operator, the Ray Data scheduler keeps a queue of ready partitions, either a `Dataset.read` input, e.g., a batch of input filenames, or an intermediate partition. We use Ray as a decentralized dataplane, so that the Ray Data scheduler only needs to keep *references* to partitions, not the physical data [99]. In a loop, the scheduler scans the current resource availability, chooses a ready partition, then passes it by reference to a new Ray task along with a description of the physical operator to execute. We describe the policy in Section 2.4.3.

We build upon and extend Ray's fault tolerance. Ray provides lineage-based recovery and stores objects in separate processes from the workers, so that individual executor failures do not impact materialized partitions [99]. If a materialized partition is lost due to node failure, Ray automatically recreates it by resubmitting the tasks in its lineage. In Section 2.4.2.1, we describe the extensions we made to Ray's recovery method to support streaming repartition.

2.4 System Design

2.4.1 Query Planning

The query planner takes as input a logical DAG and parameters such as the target maximum partition size. It generates a physical DAG, then applies a series of optimizations on the physical DAG, for example, adjacent operators that require the same resources are fused into one physical operator.

During execution, tasks may dynamically repartition the data. The initial number of partitions to use for a `read` operator is decided by the query planner. It needs to be sufficiently large to utilize all available execution slots (usually CPUs), but not so large that each partition is tiny, which can increase system overheads. It is also upper-bounded by the number of input files. By default, we aim to produce partitions that are 1—128 MB in size. We compute the initial number of partitions based on the following heuristics: the number of initial execution slots, the estimated output size of the `read` operator, and the user-requested value, if any.

Some `Dataset` consumption APIs (Table 2.2) induce transformations on the physical DAG. The `Dataset.write` call is appended to the DAG as a `map` that writes items to external storage. The `iter` API returns a stream of output records. Under the hood, this is implemented by fetching and buffering output partitions. The `iter_split` call shards the outputs into N streams, each of which can be passed to a different process. This is useful for cases such as distributed data-parallel training where the dataset is sharded among N trainers. To implement `iter_split`, the query planner launches a Ray actor before execution to coordinate the dynamic partition assignment. The stream readers fetch and buffer output partitions from this coordinator actor. Partitions are passed by reference to avoid coordinator bottleneck.

2.4.2 Query Execution

During execution, the Ray Data centralized scheduler has a global view of the executing tasks and the available resources. The scheduler repeatedly executes the following loop:

- Wait for an executing task to materialize an output partition. Tasks may produce multiple output partitions; if this was the last, then mark the task's resources as free.

- While there are free resources and ready partitions, launch new tasks using the policy described in Section 2.4.3. Mark the task's required resources as used.

The scheduler passes in the task description: (1) a closure of the physical operator to execute, (2) references to the task's input partition(s), and (3) the target output partition size. We leverage Ray to ensure that the task's input partition(s) are made local to its executor.

2.4.2.1 Streaming repartition

Although the query planner makes an initial estimate of the number of output partitions to use, this value may not be optimal. When a physical operator's outputs are much larger or smaller than its inputs, the initial partitioning will produce too-large or too-small inputs. Too-large partitions can cause out-of-memory failures from buffering many records at the executor. On the other hand, too-small partitions are also inefficient; while the Ray Data scheduler only stores references to partitions, it still must manage some metadata per partition and carry out RPCs to schedule corresponding task(s).

Ray Data introduces a technique called *streaming repartition* to handle such cases (Fig 2.3b). To support this, we extend Ray with remote *generator tasks*, which enable Ray tasks to produce a *dynamic* number of outputs, and to *pipeline* execution with the task's caller (the Ray Data scheduler). Whenever the task produces a new output, it notifies the Ray Data scheduler via RPC. Upon receipt, the scheduler can immediately launch a downstream task. Meanwhile, the upstream task can continue producing its next output partition.

Ray Data tasks take a target partition size from the scheduler and determine locally how to partition their outputs. When executing a task, the worker accumulates processed rows to a local output buffer. Once the output buffer exceeds the maximum target partition size (128 MB by default), it uses the `yield` keyword in Python to serialize and materialize the partition in Ray's object store. If a logical operator produces much less data than it consumes, a task may produce a too-small partition. To handle this, we simply coalesce partitions by having the Ray Data scheduler pass multiple partitions from different upstream tasks to a single downstream task.

2.4.2.2 Failure recovery

Ray provides automatic recovery for objects (intermediate partitions) as long as (1) the driver is alive, (2) the tasks that created them are deterministic and side effect-free [99], and (3) the task arguments and outputs are immutable. For generator tasks, (3) is no longer true, because we do not know at submission time how many outputs the task will produce. However, we note that if (2) is true, then streaming repartition can be made deterministic.

In particular, given a target partition size, a pure transform, we ensure that a Ray Data task will produce the same stream of output partitions if executed on the same input partition(s).

To support failure recovery for generator tasks, we modify Ray's recovery subsystem to handle tasks with an unknown number of outputs. Generator tasks are initially launched with an unknown number of outputs. On the first successful execution, the task's caller records the number of outputs that the task produces. If any of the task's outputs are lost, we recover by re-executing the entire task. If the task produces a different number of outputs, we throw an error.

Similar to other batch and stream processing systems [21, 69, 105, 62], if the centralized scheduler dies, Ray garbage-collects the job and it must be re-executed from the beginning. In the future, this case can be optimized through known techniques for asynchronous global checkpointing [2, 21].

2.4.3 Adaptive Scheduler

The goal of the scheduler is to minimize the job completion time while keeping the total memory usage of intermediate data below the system limit. To achieve this, the scheduler uses a principle of equalizing the processing rates of each operator (in bytes per second). Intuitively, if processing rates are not equal, slower operators will accumulate pending inputs, eventually exhausting the memory buffer. Each operator's processing rate can be controlled by deciding how many tasks to run in parallel for that operator. The processing rates are estimated online using run-time statistics, including operator task durations and average task input–output data size ratios, because these properties are difficult to predict ahead of time, and could vary depending on the actual data being processed.

2.4.3.1 Algorithm

The input to the scheduler is as follows:

- The physical DAG of operators (§ 2.4.1). Each operator can process data items in parallel tasks and is annotated with its resource requirements, e.g., `\{GPU:1\}`.
- Resource limits: the total number of CPU, GPU, or custom resource slots, and the total memory capacity of the system.

Algorithm 1 describes the scheduling loop in step (3) of Figure 2.5. From all qualifying operators, it picks the one with the least amount of data accumulated in its output buffer. The intuition is that this operator would be producing data at a slower rate than it is consumed, and thus needs more parallelism. This policy works well for equalizing the

Algorithm 1 An adaptive scheduler that equalizes operator processing rates

```
1:  Initialize budget ← totalMemoryCapacity
2:  while not all operators are done do
3:      Update resource utilization and run-time estimates
4:      Update budget                                          ▷ Algorithm 2
5:      if budget ≥ outputPartitionSize(source) then
6:          Launch task of source
7:          budget ← budget − outputPartitionSize(source)
8:      end if
9:      Q ← ∅                                                  ▷ Set of qualified operators
10:     for each operator op in DAG do
11:         if hasInputData(op) and
12:     hasAvailableResources(op) and
13:     hasOutputBufferSpace(op) then
14:             Q ← Q ∪ {op}
15:         end if
16:     end for
17:     if Q ≠ ∅ then
18:         selected ← argmin_{op∈Q} bufferedOutputsSize(op)
19:         Launch task of selected
20:     end if
21: end while
```

operator processing rates when there is enough memory to store the intermediate data while all executors are utilized.

However, when memory is constrained, the policy (without lines 4–8) will result in resource under-utilization, as seen in Figure 2.4a. The ideal solution is to keep the pipeline full and start *source* tasks, i.e. tasks for the source operator, as early as possible, as in Figure 2.4b. To achieve this, lines 4–8 add a higher-priority optimistic policy for scheduling source tasks, described further in the next section.

2.4.3.2 Input Rate Control

The input rate, i.e. the rate at which source tasks are scheduled, must approximate the pipeline's overall throughput: If the source tasks are launched too slowly, the downstream operators will have no input to process, wasting compute resources. Conversely, if the source tasks are launched too aggressively, then these tasks may starve downstream operators and eventually cause slowdowns from back-pressuring of the source operator and/or spilling to disk.

We use a dynamic *memory budget* algorithm to regulate the rate at which source tasks* are launched. Intuitively, the budget is an optimistic estimate of the memory available for new data partitions to enter the system. When a source task is launched, we deduct its estimated output size from the budget. At every second, the budget is updated using Algorithm 2, which estimates the rate at which data leaves the pipeline.

Algorithm 2 Algorithm for periodically updating the memory budget

```
P ← 0                                          ▷ Total processing time per partition
α_0 ← 1                                        ▷ α_i := Input:Output size ratio for op_i
for i ← 1 to numOps do
    E_i ← availableExecutionSlots(op_i)
    T_i ← estimatedTaskDuration(op_i)
    if op_i is not source then
        I_i ← estimatedInputSize(op_i)
        O_i ← estimatedOutputSize(op_i)
        α_i ← α_{i−1} · O_i/I_i
    end if
    P_i ← (T_i/E_i) · α_{i−1}
    P ← P + P_i
end for
budget ← budget + outputPartitionSize(source)/P
```

We will walk through the algorithm using the following example: load (CPU) → transform (CPU) → inference (GPU) pipeline, running on a cluster with 8 CPUs and 4 GPUs.

- Consider the first non-source operator: transform. Assume that the number of available execution slots to run the task is $E_1 = 6$ (out of 8 CPU slots). Assume the average task duration is $T_1 = 12$ seconds. Then the processing time of this stage is $P_1 = T_1/E_1 \cdot \alpha_0 = 12/6 \times 1 = 2$, where α_0, the output multiplier, is initialized to 1. In other words, transform takes 2s to process a source partition on average.

- Assume the transform's average output is double the size of its input, i.e. $\alpha_1 = 2$. Now consider inference. Assume the number of available execution slots is $E_2 = 4$ (GPU), and the average task duration is $T_2 = 2$ seconds. Then $P_2 = T_2/E_2 \cdot \alpha_1 = 2/4 \times 2 = 1$, i.e. the inference operator takes 1 second per source partition.

- Adding them up, $P = 2 + 1 = 3$ seconds per source partition. In other words, every $\sim$ 3s, the budget will be replenished to allow for one more source task to run.

*For DAGs with multiple sources, the launch rate for each operator should be proportional to their data output size.

If the run-time estimates of each operator's processing rates are perfectly accurate, i.e. if there is no variance in the processing rates, it can be shown that the schedule produced is optimal. However, when there is variance in processing times or output sizes, the budget algorithm could overestimate the overall processing rate. Nevertheless, the algorithm is stable because it creates a negative feedback loop. If it overestimates the pipeline processing rate, more source tasks might launch and temporarily cause backpressure, or objects in the buffer to spill to disk. However, since these tasks still occupy execution slots, they will reduce the parallelism of downstream operators and lower the replenishment rate of the budget, which in turn limits the source task launch rate. This will in turn release more resources for downstream operators to run, consuming the buffered intermediate data, and bringing the pipeline throughput back to equilibrium.

Ray Data also provides a conservative scheduling policy in which a task is launched only when its output space in the memory can be guaranteed, similar to Figure 2.4a. This policy enforces a hard memory limit and never spills, albeit at the risk of under-utilizing of executor slots when memory is limited.

2.4.4 Implementation

Current batch and stream processing systems could in principle be modified to use the streaming batch model, but would require fundamental changes to their execution models. For example, Spark would need to support pipelined stage execution, while Flink would need to support zero-downtime reconfiguration. We choose to implement Ray Data on top of Ray because Ray exposes a lower-level execution model based on dynamic task execution. Ray provides powerful features such as automatic data movement, lineage-based recovery [99], and automatic disk spilling [51]. This makes it convenient to build centralized schedulers like Ray Data while leveraging Ray as a decentralized dataplane. However, Ray also treats the task logic, inputs, and outputs as black boxes. Thus, it is difficult to directly extend Ray with data processing-specific features such as dataset partitioning, streaming repartition, and pipeline-aware task scheduling. Instead, we implement Ray Data as a Ray library, which allows us to build such features with minimal changes to the Ray core. Ray Data is written in $\sim$ 90K Python LoC, including $\sim$ 45K LoC for the query planner, scheduler and executor logic.

2.5 Evaluation

We evaluate a range of heterogeneous workloads with a focus on multimodal batch inference and training. We use benchmarks taken from the MLPerf suite [53, 78], and supplement with additional image-to-image, video-to-video, and video classification benchmarks. We aim to answer:

- § 2.5.1: How does the streaming batch model compare to traditional batch or stream processing models when running heterogeneous ML workloads, in terms of throughput and adaptivity vs. fault tolerance?
- § 2.5.2: How does distributed and heterogeneous execution improve on throughput per dollar compared to ML-specific single-node data loaders?
- § 2.5.3: How well do all systems adapt to memory pressure in heterogeneous settings, and what are the system overheads?

We compare the following systems:

- Batch processing (Figure 2.2a): Apache Spark 3.5.1. Spark executors are tied to resources: if one stage requires 1 CPU/task and another 1 GPU/task, then each executor acquires 1 CPU+1 GPU, greatly reducing throughput when there are fewer GPUs than CPUs. For fairer comparison, we specify 1 CPU/task and manually schedule GPU tasks by repartitioning the "GPU" stage to the number of GPUs available. This maximizes parallelism, at the cost of Spark's main advantages: reconfigurability and fast recovery.
- Stream processing (Figure 2.2b): Apache Flink 1.19.0. We manually tune Flink's parallelism per operator to the highest possible without running out of memory.
- Single-node stream processing: tf.data [62]. A data loader for ML training. tf.data uses multithreading and can reconfigure the number of threads per operator.
- Single-node stream processing: PyTorch DataLoader (PyTorch DL) [73]. Similar to above, but uses multiprocessing and requires manual fusing of all operators.
- Streaming batch: Ray Data (implemented over Ray 2.40.0), codenamed Radar in the figures. We also modify Ray Data to emulate batch processing (Radar-staged) and stream processing (Radar-static), for apples-to-apples comparison of the execution models in § 2.2. Radar-staged materializes each stage before starting the next, while Radar-static sets a static parallelism per operator and disables the adaptive scheduler described in § 2.4.3.

For training workloads, we compare only against tf.data and PyTorch DataLoader because these systems were custom-built for data preprocessing for training. They do not support distributed execution and target CPU-only preprocessing for a co-located GPU trainer, making it difficult to run batch inference, which alternates between CPU and GPU stages. Thus, we use Spark and Flink as comparisons for batch inference.

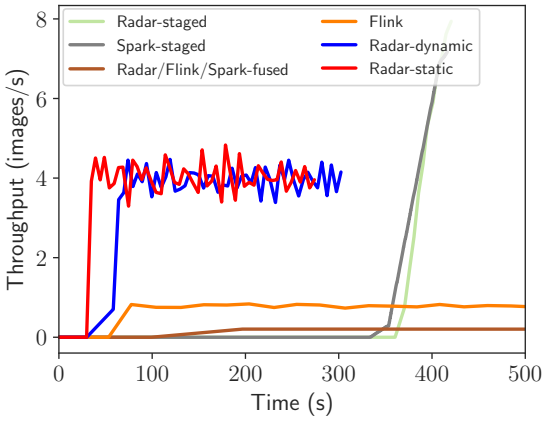


Figure 2.6: Image generation throughput comparison. Radar-dynamic (blue) can match the performance of hand-tuned Ray Data-static (red) with static parallelism, and outperform other systems. *-staged execute synchronously; no results are available until the last stage begins at $\sim$ 330s.

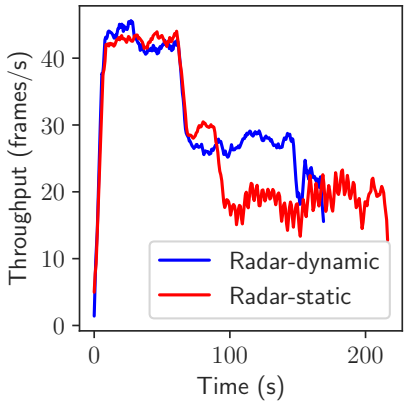


Figure 2.7: Video generation throughput comparison. Radar-dynamic (blue) achieves 28% better throughput compared to Radar-static (red) with static parallelism.

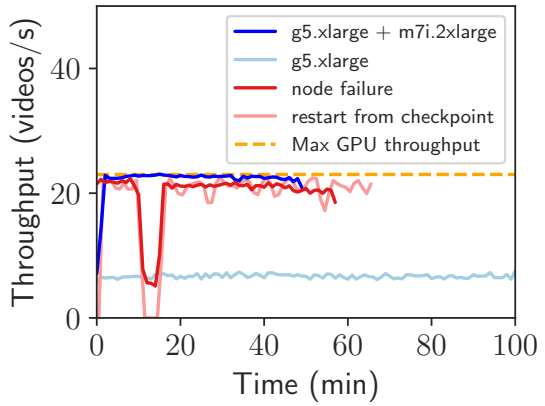


Figure 2.8: Fault tolerance comparison. Ray Data can handle isolated executor failures (blue) with negligible impact on throughput, and node failure (red) without restarting the job. The emulated checkpoint-and-restore (pink) leads to job downtime and longer completion time due to recomputation.

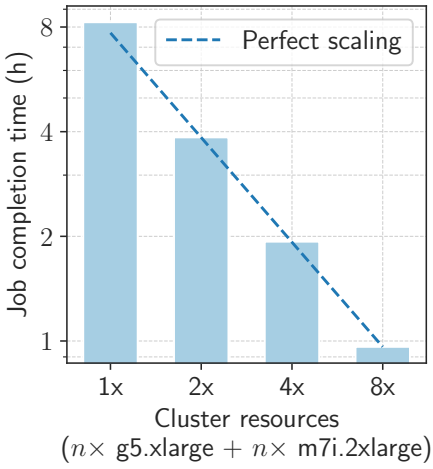


Figure 2.9: Scalability analysis. Ray Data is able to achieve strong scaling with respect to the number of nodes.

2.5.1 Inference: Comparison to distributed batch and stream processing

2.5.1.1 Image-to-Image Generation

Image-to-image generation uses a generative model to produce a new image from a source image and prompt. Listing 1 consists of the steps: (1) `read_images`: download the InstructPix2Pix [18] image dataset from S3, (2) `decode`: decode JPEG images to tensors, (3) `preprocess`: normalize tensors, (4) `Img2ImgModel`: use Stable Diffusion [80] to generate a new image based on the input image and prompt, and (5) `encode_and_upload`: Encode to JPEG and upload to S3.

Listing 1 Image-to-image generation expressed in Ray Data API.

```
ray.data.read_images(INPUT_PATH).map(decode).map(preprocess)
  .map_batches(Img2ImgModel, batch_size=B, num_gpus=1)
  .map_batches(encode_and_upload, batch_size=B)
```

Figure 2.6 shows throughput over time of the different execution models on 1 g5.2xlarge VM (8 vCPU, 1 A10G GPU). The *-fused baselines fuse all operators. This avoids materialization but limits overall parallelism to the scarcest resource, in this case 1 GPU. The resulting throughput is thus the lowest, confirming that operator fusion is undesirable on heterogeneous resources. Spark-staged and Ray Data-staged disable fusion of CPU and GPU operators to decouple their parallelisms. Stages execute synchronously, so no results are available until the last stage begins at $\sim$ 330s. The overall throughput is also significantly lower because all intermediate stage results need to be materialized and in this case spilled to disk.

For Flink, we also disable fusion, by setting parallelism to 8 and 1 for CPU and GPU operators, respectively. Flink uses multithreading to share each CPU among its multiple operators (`read+decode+preprocess` vs. `encode\_and\_upload`). Execution is pipelined across physical operators, and only a fraction of intermediate records are materialized at once, so Flink can produce results almost immediately. However, there is significant overhead due to serialization of image data between the Python UDF and the Java-based Flink. Thus, Flink's throughput is 70% lower than Ray Data's.

Because of Flink's serialization overheads, we use Radar-static to emulate a stream processing baseline. We manually determine the best static parallelism per operator before execution. This achieves the best throughput, at 4 images/s. Radar-dynamic is the default system, with the adaptive scheduler (§ 2.4.3). This automatically converges to the same throughput as Radar-static.

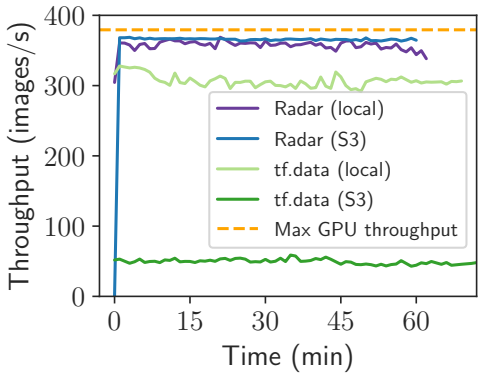


Figure 2.10: Training ResNet-50.

	Resources	Images/s	Run time (hours)	Total cost
PyTorch DL (stream)	4× p4de.24xlarge	2,811	111.3	$18,192
Ray Data-staged (batch)	4× p4de.24xlarge	0, then 4,068	90.3 **(-19%)**	$14,753 **(-19%)**
Ray Data (streaming batch)	4× p4de.24xlarge 40× g5.2xlarge	4,075	76.8 **(-31%)**	$16,275 **(-11%)**

Table 2.3: Run time and cost for one epoch of Stable Diffusion pre-training.

Takeaways: (1) Stream and streaming batch models outperform batch models for heterogeneous pipelines, due to asynchronous stages and lower peak memory footprint, (2) Ray Data's adaptive scheduler matches the best stream processing baseline (Radar-static), with no configuration needed.

2.5.1.2 Video-to-Video Generation

Video-to-video generation uses a generative model for use cases including super-resolution, interpolation, or de-blurring [22, 43, 100]. Compared to § 2.5.1.1, video generation adds significant memory pressure: each video decodes to many frames. Also, high workload variability is common due to varying video lengths, resolutions, and encodings. Thus, we use this workload to evaluate the system's ability to adapt compute and memory allocations based on real-time usage.

We run a 3-stage pipeline on a g5.2xlarge VM: (1) *download+decode* (CPU): Download the YouTube-8M dataset [1] from S3. Video resolution ranges from 320p to 720p, with lengths 2–5 minutes, encoded using H.264. Each video is decoded into multiple 128-frame sequences, the maximum batch size allowed by GPU memory. Earlier videos are low-resolution. (2) *generate* (GPU): Run the RealBasicVSR [22] super-resolution model to produce HD frames.

(3) *encode+upload* (CPU): Encode the HD frames using H.264, then upload the videos to S3.

Figure 2.7 compares the throughput over time of Radar-static and Radar-dynamic, the two best systems in § 2.5.1.1. Radar-static emulates stream processing and allocates the pre- and post-processing stages of 4 CPUs each. Throughput is initially high but later drops because this initial configuration is not optimal for the later high-resolution videos, which are more memory-intensive and take longer to download and decode. Meanwhile, with Radar-dynamic, the Ray Data scheduler dynamically re-balances tasks according to task duration and memory usage, achieving 28% better throughput.

Takeaway: Ray Data adapts quickly to changing workloads because tasks can run on any executor and the adaptive scheduler can dynamically reallocate resources.

2.5.1.3 Fault tolerance in heterogeneous clusters

We first demonstrate Ray Data's ability to scale with heterogeneous clusters. We run the VideoMAE [92] for video classification, on the Kinetics-700-2020 dataset with 635,000 videos and 871 GB in size. This workload is similar to that in § 2.5.1.2 except with less postprocessing. Data is stored on Amazon S3. Figure 2.8 shows the throughput over time when processing 10% of the dataset on 1 g5.xlarge node (4 vCPU, 1 GPU). The single-node configuration produces 6.2 videos/s, and is bottlenecked by CPU preprocessing. Ray Data can scale data preprocessing independent of GPU inference. We launch a m7i.2xlarge node (8 vCPU) to create a heterogeneous Ray Data cluster with 12 vCPUs and 1 GPU. Overall throughput then reaches 21.7 videos/s, or 94% of the maximum GPU throughput. The overall job cost is also 60% less, as CPU-only nodes are more cost-efficient. We further test scaling the cluster up to 16 nodes, or 8 GPUs and 96 vCPUs in total. Figure 2.9 shows that Ray Data is able to achieve near-perfect strong scaling.

Fault tolerance. CPU executor failures are common in heterogeneous clusters mostly due to preprocessing tasks running out of memory. Ray Data can transparently recover from isolated executor failures without interrupting the job. To further evaluate the cost of fault recovery, we intentionally fail the CPU worker node after 10 min, and reconnecting it after another 1 min. We compare Ray Data's native lineage reconstruction failure recovery against checkpointing, which is commonly used in stream processing systems. Figure 2.8 shows that Ray Data's throughput drops to that of the remaining node during worker node failure, and restores when the node rejoins without interrupting the job. In comparison, we implement a checkpointing-based recovery, by saving progress every 5 min, then restarting the job to load from the last checkpoint when failure occurs. As expected, the system makes no progress until $t = 18$min, due to having to restart the job and perform redundant work.

Takeaways: Ray Data enables scaling with heterogeneous clusters to reduce overall cost by 60%, despite using more nodes. Compared to stream processing systems that use global checkpointing, failures at the additional CPU-only nodes have little impact on end-to-end throughput, allowing further cost reduction via spot instances.

2.5.2 Training: Comparison to ML data loaders

2.5.2.1 ResNet Training

We run the ResNet-50 ImageNet training benchmark from MLPerf [53]. The data preprocessing pipeline loads images from local disk (local) or cloud storage (S3), decodes, and randomly crops and flips the images. We compare training throughput of tf.data vs. Ray Data on a g5.2xlarge VM. We do not measure PyTorch DataLoader, as tf.data showed comparable or better results for the same benchmark in [62].

Figure 2.10 shows training throughput over time. tf.data executes data preprocessing using a pool of worker threads running in each GPU trainer process. Thus, the job fate-shares with *any* preprocessing task that fails due to out-of-memory (OOM). When reading data from local disk, tf.data's throughput is 19% lower than Ray Data's because a lower batch size was required to prevent OOM failures. Meanwhile, Ray Data is able to complete because GPU trainer failures are isolated from CPU worker failures, and CPU workers can be respawned in seconds, without impacting pipeline throughput (§ 2.5.1.3).

When reading data from S3, tf.data is 88% slower than the max GPU throughput because S3 loading is the bottleneck. Meanwhile, Ray Data can use heterogeneous clusters to scale out S3 loading independent of the GPU trainers. By adding a m7i.2xlarge node for data preprocessing, the overall training throughput reaches 93% of the max GPU throughput.

Takeaways: Compared to single-node ML data loaders, Ray Data offers: (1) failure isolation between heterogeneous resources, and (2) ability to leverage heterogeneous clusters.

2.5.2.2 Pre-Training Stable Diffusion

Pre-training of large ML models is one of the most demanding heterogeneous workloads. We run the Stable Diffusion (SD) pre-training pipeline shown in Figure 2.1b and compare different execution modes. We execute 1 training epoch over a dataset of 2 billion images, on a cluster of 4× p4de.4xlarge nodes, with 8 A100 GPUs on each node. This pipeline is challenging because it requires both CPUs and GPUs for data preprocessing: (1) `loadText/loadImage+clip` (CPU): Load pairs of image and text, perform preprocessing, (2) `Encoder` (GPU): Use a pre-trained encoder model, one for images and one for text, to produce dense embeddings, and (3) `UNet.train()` (GPU): Train SD on the embeddings.

Table 2.3 shows training throughput and total cost. PyTorch DL is a data loader custom-built for PyTorch that statically partitions work on a process pool. Its throughput is lowest because `Encoder` preprocessing competes with trainers for GPU memory. Ray Data-staged emulates batch processing by running data preprocessing as an offline job and storing pre-computed embeddings in cloud storage. This is preferable if the same embeddings are used multiple times. Ray Data-staged achieves 19% higher throughput because `UNet` is given full GPUs.

Ray Data runs all data preprocessing concurrently with training. This is preferable if using random transforms or for iterative development. Ray Data also leverages heterogeneous clusters, placing `Encoders` on smaller A10G GPUs (g5.2xlarge). This results in 31% better throughput than PyTorch DL, because `UNet` has full GPU resources, and 15% better throughput than Ray Data-staged, because embeddings are kept in memory.

Takeaways: Compared to existing ML data loaders, Ray Data can: (1) be used for both batch and online data preprocessing, and (2) leverage heterogeneous distributed execution.

2.5.3 Microbenchmarks

2.5.3.1 Memory-aware scheduling

We evaluate how batch, stream, and streaming batch systems schedule heterogeneous pipelines with and without memory pressure. The 3-stage pipeline is: (1) Load (CPU): 160 tasks, each producing 500 1 MB rows after 5s, (2) Transform (CPU): sleep for 0.5s per row, then return a different 1 MB row, (3) Inference (GPU): 0.5s per batch of 100 rows. We use 1 m6i.2xlarge node with 8 vCPUs, 4 simulated GPU slots, and 32 GB RAM. The theoretical best job completion time with unlimited memory is $(160 \times 5s + 800 \times 0.5s)/8 = 150s$.

Figure 2.11 shows job completion time vs. total memory limit. We limit memory through system-specific configurations, e.g., executor memory for Spark. We also use POSIX `rlimit` to verify that each system respects its memory limit, and tune each system's parallelism (e.g., executor count) if not.

Spark materializes all data between stages, achieving at best 2.35× optimal run time. At 12–14GB memory, Spark must use fewer executors resulting in 4.34× the optimal run time, and at lower memory limits, Spark is unable to finish. This is because Spark requires static partitioning (§ 2.2.2), and the initial number of `Load` tasks produces too-large partitions.

Flink is less sensitive to the memory limit than Spark and achieves up to 1.68× optimal. This is because executors dynamically materialize output partitions to avoid running out of memory (§ 2.2.3). At lower memory limits, Flink must run fewer executors because it uses multithreading and slot sharing to multiplex a CPU slot among physical operators,

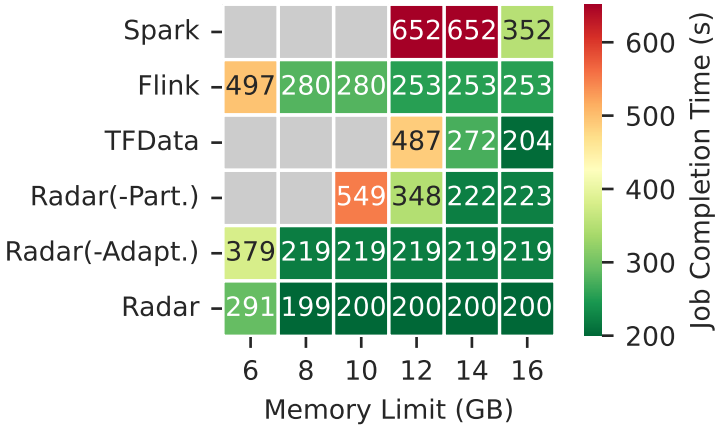


Figure 2.11: Synthetic benchmark run times for systems under different memory limits. Grey means the system is unable to finish due to OOM. Radar(-Part.) means Ray Data without streaming repartition. Radar(-Adapt.) means Ray Data without adaptive memory-aware scheduling.

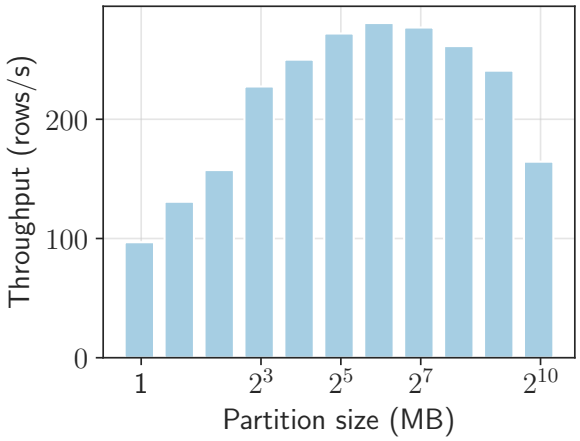


Figure 2.12: Effect of partition sizes on throughput in Ray Data.

making executors vulnerable to OOM under memory pressure. This results in up to 2× worse throughput.

We also compare against tf.data because unlike PyTorch DL, it offers an adaptive scheduler and memory budget, similar to Ray Data. However, we found that the memory budget was not always enforced, requiring manual tuning of the thread count. tf.data achieves the same throughput as Ray Data at 16 GB memory limit, but is unable to finish at lower memory limits.

Ray Data is able to finish in 1.3× the optimal run time at all memory limits except

the lowest, due to Ray Data's streaming repartition (Figure 2.3b) and adaptive scheduler (§ 2.4.3). We further conduct ablation studies on Ray Data's performance. Radar(-Part.) disables Ray Data's streaming repartition, resulting in too-large initial partitions similar to Spark. Radar(-Adapt.) disables Ray Data's adaptive scheduler, resulting in a conservative policy similar to Figure 2.4a and 10–88% worse performance than Ray Data. Ray Data is also less sensitive than Flink to memory pressure because the system explicitly time-slices executors at task granularity, instead of using multithreading.

Takeaways: For heterogeneous applications under memory pressure, batch processing systems are unstable. Ray Data is as stable as Flink, due to its streaming repartition, and also more adaptive, thanks to its scheduler.

2.5.3.2 Overhead of partitioning

Compared to stream processing, Ray Data also uses dynamically sized partitions, but with a centralized scheduler. A possible concern is the system overhead per partition. We evaluate the impact of partition number on throughput in Ray Data with a 2-stage synthetic pipeline. We use 8192×1 MB input rows, and simulate 10 ms processing time per row per stage. Figure 2.12 shows the throughput vs. partition size. The smallest partition sizes incur overhead from RPCs and bookkeeping, and the largest result in poor load-balancing. To strike a balance, Ray Data's default target partition size is 128 MB.

2.5.3.3 Streaming repartition

We run the video batch inference workload (§ 2.5.1.3) on a single video file with 4,500 frames, and simulate 0.5 s preprocessing time per 16-frame batch. We run on a g5.2xlarge VM with 8 vCPUs.

In video processing, a single video contains thousands or more frames, each of which requires downstream processing. Without repartitioning, the available downstream task parallelism is limited to the number of video files. As shown in the bottom part of Fig 2.13, this makes `preprocess` the bottleneck, leaving the GPU utilization under 10%.

With streaming repartition (§2.4.2.1), as the `read+decode` task runs, Ray Data creates small batches of decoded video frames, and passing them to the downstream operator in a streaming fashion. This reduces the memory required for intermediate outputs, and enables the `preprocess` stage to run at a higher parallelism equal to the number of CPUs. This increases the GPU utilization to 35%, reducing the job completion time by 6×.

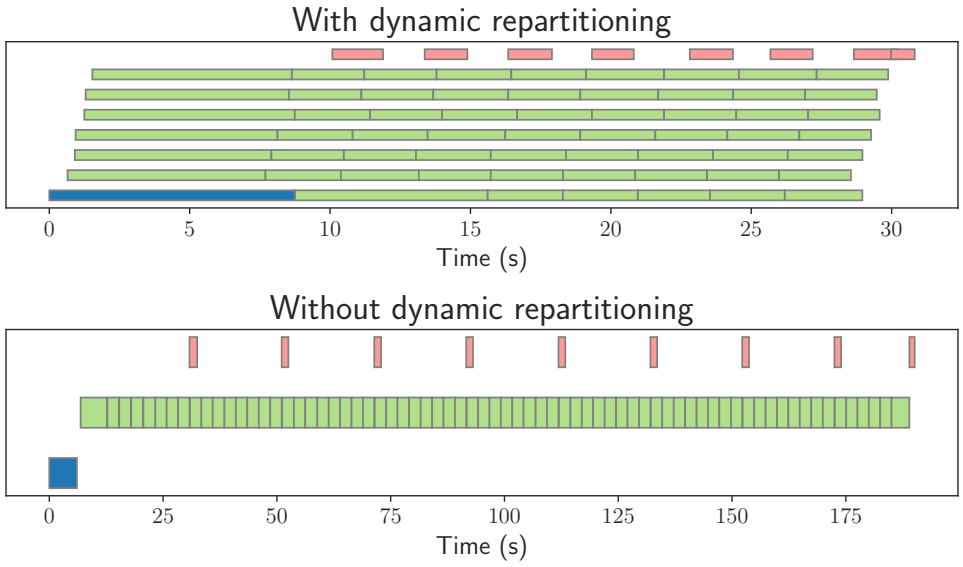


Figure 2.13: Comparing execution schedules with vs. without dynamic repartitioning. Legends: `read+decode`; `preprocess`; `predict` (GPU).

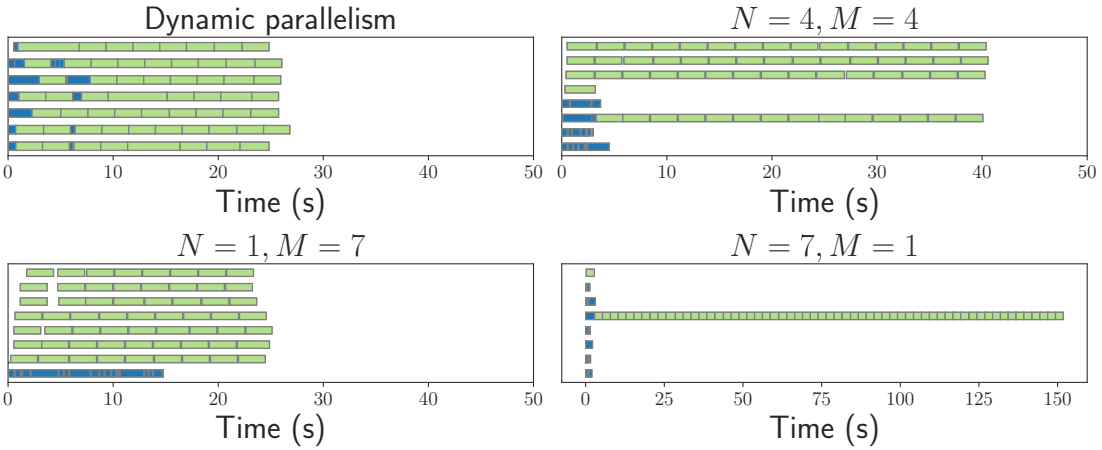
2.5.3.4 Fixed vs. dynamic parallelism

We run two variations of the workload to demonstrate the adaptability of the Ray Data scheduler. In the first scenario (Fig 2.14a), `preprocess` takes longer than `read+decode`. In the second scenario (Fig 2.14b), `read+decode` is simulated with an additional 1s latency per video file. In both versions, we run the workload on 20 videos, with different fixed values of `read+decode` and `preprocess` parallelisms. Figure 2.14 shows that there is no single fixed setting[†] that can yield desirable execution schedules in both versions. Meanwhile, Ray Data (top-left of Fig 2.14a and 2.14b) produces the fastest execution schedules in both scenarios by varying the parallelism of the two operators throughout the timeline. This contrasts with typical stream processing systems, in which operator parallelisms must be decided ahead of time, making the system less flexible and performant for dynamic workloads.

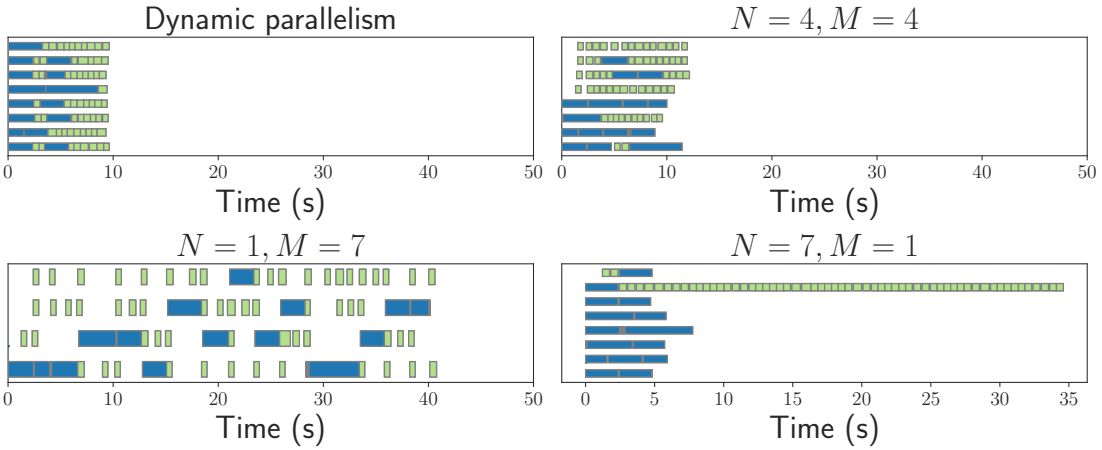
2.5.3.5 Fractional parallelism

In this microbenchmark, we demonstrate that the streaming batch model can maximize the resource utilization when fractional parallelism is required. Consider a two-stage data pipeline, in which the first stage takes 1 second on average, and the second stage takes 2 seconds. Ideally, the operator parallelisms should be set as $2:1$ to balance the throughput. In traditional stream processing systems such as Flink, this is unattainable on a 8-CPU

[†]For example, $N = 1, M = 7$ works well in Fig 2.14a, but not in 2.14b.



(a) A scenario when `preprocess` is slower.



(b) A scenario when `read+decode` is slower.

Figure 2.14: Comparing execution schedules with fixed vs. dynamic parallelisms. Each row represents one execution slot. N is `read+decode` parallelism; M is `preprocess` parallelism. In both cases, dynamic parallelism produces the optimal schedule.

machine, because it requires setting the operator parallelisms to be 2.67 and 5.33, respectively. Since these systems allocate executors to operators statically, they cannot support fractional parallelism. In contrast, the streaming batch execution model allows Ray Data to multiplex executors for both stages dynamically during run time. The Ray Data scheduler can dynamically start a task for either stage in order to balance the throughput, effectively achieving a parallelism ratio of 2 : 1 over time. Figure 2.15 shows that when comparing to a static allocation of 4–4 executors for each stage, the dynamic allocation increases the utilization of the execution slots, manifested as fewer bubbles in the schedule, and 19% faster job completion time.

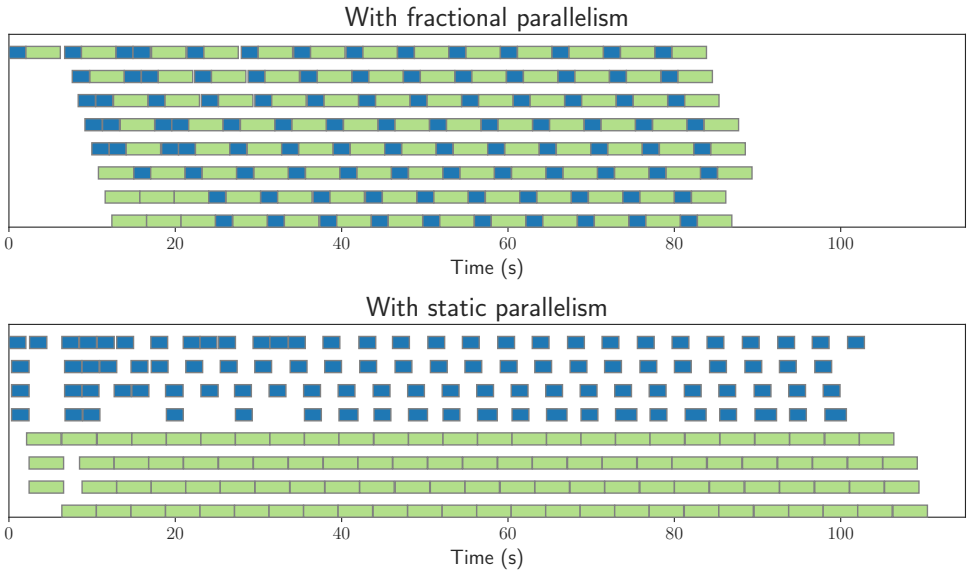


Figure 2.15: Dynamically allocated executor slots can achieve fractional parallelism with better resource utilization (fewer bubbles).

2.5.4 Solver for Discrete-time Scheduling

To verify the efficacy of the online scheduling algorithm, we develop a discrete-time simulation environment, in which tasks have fixed execution times, and a discrete-time solver that can find the optimal schedule to run a data pipeline, subject to specified resource constraints.

The input to the solver is a data pipeline, the total data size, and the resource constraints. The data pipeline is described as a chain of operators. Each operator processes data in tasks. Each task has an input size and an output size measured in number of partitions, and we assume each task has a known duration. Each task also has a resource requirement, e.g. 1 CPU or 1 GPU.

The total data size is also measured in number of partitions. The resource constraints describe how many execution slots are available for each resource type (CPU or GPU), and also has a memory buffer limit, indicating how many intermediate partitions in total can be stored in the temporary memory buffer.

Finally, the solver has a length limit, measured in time ticks, for any solution returned. This is such that the solution space is bounded.

2.5.4.1 Algorithm

The solution space is defined by the set of all possible execution states. The execution state consists of:

- Time since the start.
- The state of each executor, i.e. which operator task is running.
- The state of the shared memory buffer, which is the number of partitions stored in the buffer.
- The state of each operator, which is the number of pending tasks.

The solver starts from the initial state (time 0, all executors idle, buffer empty, and all tasks pending). For each state, it generates the next state by emulating the execution: advancing the tick, updating executor states, updating the progress of running tasks, updating the memory buffer, etc. The number of next states is determined by the size of the set of all possible scheduling actions, which is the power set of all possible scheduling *primitives*. A scheduling primitive would be "schedule the next task operator i onto executor j."

The solver runs a variation of the A* search algorithm to try to arrive at the first completion state (in which no more tasks are pending). In the priority queue, the states are sorted by the number of completed tasks, i.e. it prioritizes states that make further progress. The solver returns the optimal job completion time after all possible states are visited.

The naive search algorithm is not practical due to its high time complexity ($O((E \cdot T)^N)$), where N is the total number of tasks, E the total number of executors, and T the time limit). We use the following optimizations to bring the complexity down to $O(2^N \cdot T)$, making it more practical for large scheduling problems.

- Symmetry of tasks and executors. We assign a canonical ordering of the executors, i.e. the first task always starts on the lowest-numbered executor. This gets rid of a large class of duplicate states, where the task timings are the same, except that they run on different executors.
- Temporal equivalence. We notice that the optimal job completion time given an execution state at time t is the same, regardless of its execution history before t. This means all states that arrive at the same task progress at time t are equivalent. This is crucial for reducing the number of duplicate states, and in many cases, reduces the problem to polynomial time.

For the scheduling microbenchmark in §2.5.3.1, the solver finds the optimal schedule with a total run time of 153 seconds.

2.6 Related Work

Unifying batch and stream processing systems. Recent efforts to unify the batch and stream processing models include Apache Beam [8] and Google Cloud Dataflow [3]. They focus on providing a unified API layer, rather than unifying the *execution model*. Attempts at execution model unification include Naiad [60], Flink's batch execution mode [21], and Spark Streaming [105]. Naiad shows that the stream processing model is suitable for producing results both incrementally and in bulk, but it does not support dynamic parallelism reconfiguration. Like other batch processing systems, Flink's batch execution mode executes one stage at a time.

Spark Streaming partitions the input stream into discretized "microbatches", each executed as a distinct Spark job. Drizzle [94] improves on Spark Streaming's latency without sacrificing adaptability, but suffers the same flaws for heterogeneous pipelines: pipelining across heterogeneous resources would require a custom inter-job Spark scheduler, and the data partitioning within a microbatch is static. Also, since Spark requires stateless tasks, it imposes high overheads in ML inference and training from repeatedly loading the model into GPU memory for each task. Petastorm [37] is a Python library that bridges Spark's data processing capabilities to ML training frameworks. It shares the same limitations as Spark.

MillWheel [2] is a stream processing system that offers efficient reconfiguration and failover by combining decentralized physical logging with a centralized load-balancer off the critical path. It offers sophisticated APIs for real-time processing, including timers, watermarks, etc. In contrast, Ray Data targets offline processing, uses lineage-based recovery to avoid data logging, and the centralized scheduler dispatches *all* tasks for a global view and finer control over resources.

Other systems have explored building distributed data processing frameworks on top of task-parallel systems such as Ray [59, 99], CIEL [61] and Dask [79]. However, all of these systems are CPU-centric and do not consider memory or heterogeneous compute.

Scheduling for resource heterogeneity. The scheduling problem described in §2.4.3 is most similar to the generalized processor sharing [67] problem. Our solution is inspired by the weighted fair queueing algorithm [14, 27]. The differences are (1) the flows in network scheduling are independent of each other, whereas operators in a data pipeline have dependencies, (2) multiple resource types, and (3) the packet processing time is usually fixed, whereas data operator processing times are unpredictable. (1) is important in that operators may produce significant intermediate data, which may not be released until the downstream operator executes.

Recent scheduling works attempt to adapt fair queuing and autotuning to heterogeneous resource environments. Dominant resource fair queuing [35] addresses the problem of (2) but not (1) or (3). tf.data [62] introduces an autotuning algorithm that uses gradient descent to

find the parallelism for each operator that reduces end-to-end latency. We instead aim to maximize overall throughput; the partition size may be used to adjust end-to-end latency.

Streaming data loaders for ML. Both PyTorch [69] and TensorFlow (tf.data [62]) provide data loaders optimized for map-style transforms for ML training. tf.data uses multithreading while PyTorch DataLoader uses multiprocessing. tf.data automatically and deterministically shards the dataset, while PyTorch DataLoader requires the user to shard the dataset themselves. However, both are single-node systems colocated with a GPU trainer and share similar limitations: they cannot execute multi-node, dataset sharding must be done before execution, and the training job fate-shares with the data loader.

Cloud-native streaming data loaders include MosaicML Streaming [44] and DeepLake [40]. The common innovations in these libraries are a specialized data format optimized for tensor storage and querying, and streaming data loading directly from cloud storage. They use Python multiprocessing for execution, which can be swapped out for Ray Data. This would decouple data loading from ML training/inference processes, thus allowing more flexible control of CPU parallelism and memory buffers and scale-out to heterogeneous nodes.

2.7 Discussion

One benefit of building Ray Data as a Ray library rather than a monolithic system such as Spark or Flink is that since the Ray core is in C++, Ray Data can support many frontend languages, including the lingua franca for ML data loading and transformations, Python. In contrast, systems built on non-native languages show high overheads when a Python frontend is used (e.g., Flink in § 2.5.1.1). Second, modifying the Ray Data scheduler is convenient, as it is written in the frontend and does not require re-compiling Ray [51].

Ray Data enables another key opportunity in future data processing systems: dynamic query planning. In this work, we present an online scheduler, but still make certain planning decisions statically, including the number of input partitions (§ 2.4.1), and the user specifies the initial cluster shape. We envision a fully autotuning and autoscaling system that can cohesively re-plan the application and resize the cluster.

Conclusion. With the rise of large language models, even modalities such as text that are traditionally not compute-intensive to load and process may now require expensive deduplication [64], GPU-based embedding computation, and joins with image or video data (Figure 2.1b). In addition, as the parallelism strategies used in inference and training pipelines become more complex, future data processing systems must also support more flexible APIs for sharding and sharing data.

In general, we believe that ML systems will continue to grow in the complexity of their data processing needs, as evidenced by trends such as test-time training [87, 34], retrieval-augmented generation [47], and multimodal models [88, 66]. To keep up with this demand, we must build more flexible, heterogeneity-aware, and scalable data processing systems.

Chapter 3

Exoshuffle: An Extensible Shuffle Architecture

The streaming batch model introduced in Chapter 2 is effective at running heterogeneous data pipelines consisting of primarily map-style transformations. However, for complex operations with data dependencies across partitions, such as sorting, grouped aggregations, and joins, fully streaming execution may not be possible.

In this chapter, we study one of the most expensive communication primitives in distributed data processing: *shuffle*. Shuffle refers to the all-to-all communication pattern in a distributed data processing system, and is notably difficult to run efficiently at scale. Prior work addresses the scalability challenges of shuffle by building monolithic shuffle systems. These systems are costly to develop, and they are tightly integrated with batch processing frameworks that offer only high-level APIs such as SQL. New applications, such as ML training, require more flexibility and finer-grained interoperability with shuffle. They are often unable to leverage existing shuffle optimizations.

We propose an extensible shuffle architecture. Exoshuffle is a library for distributed shuffle that offers competitive performance and scalability as well as greater flexibility than monolithic shuffle systems. We design an architecture that decouples the shuffle control plane from the data plane without sacrificing performance. We build Exoshuffle on Ray, a distributed futures system for data and ML applications, and demonstrate that we can: (1) rewrite previous shuffle optimizations as application-level libraries with an order of magnitude less code, (2) achieve shuffle performance and scalability competitive with monolithic shuffle systems, and break the CloudSort record as the world's most cost-efficient sorting system, and (3) enable new applications such as ML training to easily leverage scalable shuffle.

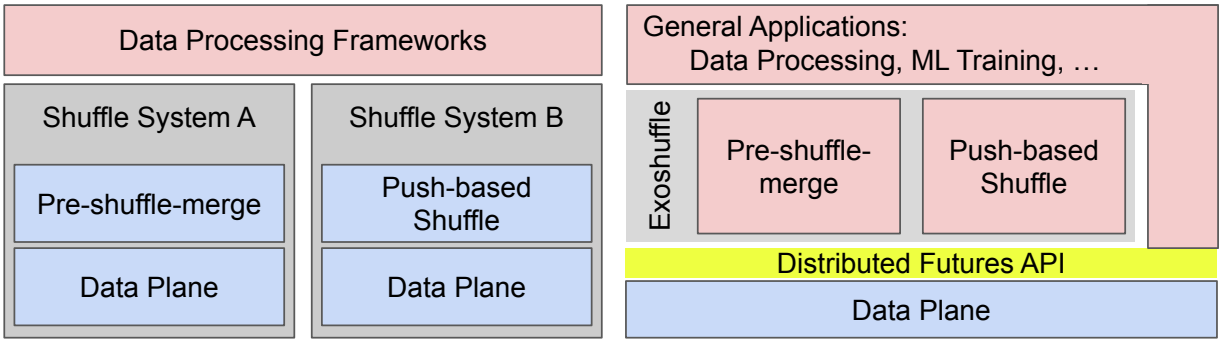
3.1 Introduction

Shuffle is a fundamental operation in distributed data processing systems. It refers to the all-to-all data transfer from mappers to reducers in a MapReduce-like system [26]. Shuffle is one of the most expensive communication primitives in these systems and is difficult to scale. Scaling shuffle requires efficiently and reliably moving a large number of small blocks from each mapper to each reducer across memory, disk, and network. It requires both high I/O efficiency, and robustness to failures and data skew. Furthermore, as the data size increases, the number of shuffle blocks grows quadratically, making shuffle the most costly operation in some workloads.

The difficulty of scaling shuffle has inspired many solutions from both the industry and research community. These shuffle implementations improve the performance and reliability of large-scale shuffle by optimizing I/O in different storage environments, such as HDD, SSD and disaggregated storage [75, 110, 84, 11]. Since performance at scale is a priority, these prior solutions are built as monolithic shuffle systems from scratch using low-level system APIs. However, these systems are costly to develop and integrate. For example, each cloud provider has to build proprietary services to support shuffle on their own storage services [11, 4, 85]. Magnet, a push-based shuffle system for Spark [84], took 19 months between publication and open-source release in the Spark project [29] because it required significant changes to system internals [83].

Furthermore, existing shuffle systems only work with batch processing frameworks, which offer high-level abstractions such as SQL or dataframe APIs. Most are synchronous in nature: the results are available only after the entire shuffle operation completes. This poses challenges for applications that require *fine-grained* integration with the shuffle operation to improve their performance by processing data as it is being shuffled, i.e., pipeline data processing with the shuffle operation. For example, ML training often requires repeatedly shuffling the training dataset between epochs to improve learning quality [57, 56]. Doing this efficiently requires fine-grained pipelining between shuffle and training: ML trainers should consume partial shuffle outputs as soon as they become ready. Today's ML developers are faced with two undesirable choices: (1) they either rebuild shuffle from scratch, once again dealing with the performance challenges of large-scale shuffle, or (2) interface with existing shuffle systems through the synchronous APIs: the shuffle results can only be consumed after all partitions are materialized, leaving pipelining opportunities on the table.

To simplify the development of new shuffle optimizations targeting different environments, and to provide fine-grained pipelining for new applications, we propose an *extensible* architecture for distributed shuffle that enables flexible, efficient, and scalable implementations. Unlike previous solutions built as monolithic systems (Fig 3.1a), we propose building distributed shuffle as a library (Fig 3.1b). Such an architecture allows: (1) shuffle builders to easily develop and integrate new shuffle designs for new environments, and (2) a broader set of applications to leverage scalable shuffle in a more flexible manner.



(a) Monolithic shuffle systems.

(b) Exoshuffle.

Figure 3.1: Exoshuffle builds on an extensible architecture. Shuffle as a library is easier to develop and more flexible to integrate with applications. The data plane ensures performance and reliability.

How can we implement shuffle at the application level (as a library) while providing high performance? To answer this question, we first identify the optimizations in past shuffle systems that are key to performance and reliability. (1) Coordination: managing the timing and placement of mapper and reducer tasks, and implementing optimizations such as merging intermediate shuffle blocks. (2) Efficient data transfer: pipelining I/O with computation to maximize throughput, and spilling data to disk to accommodate larger-than-memory datasets. (3) Fault tolerance: guaranteeing data is reliably transferred to reducers via retries or replication.

Our key observation is that we can split these optimizations between a control and a data plane. Optimizations for coordinating shuffle are implemented by the *control plane* at the application layer, while the *data plane* provides efficient data transfer and fault tolerance at the system layer. This enables developers to easily implement a variety of shuffle solutions at the application layer, while having the underlying system handle efficient data transfer and fault tolerance.

The next question is what interface should the data plane provide to the application. Our answer is *distributed futures*, an extension of RPC that allows referencing data objects in distributed memory. It allows the caller of a remote task to pass objects *by reference*, regardless of their physical locations, thus decoupling remote task invocations from physical data transfers, the latter implemented by the data plane. Distributed futures can also be passed before the data object is created, allowing the system to parallelize remote calls and pipeline data transfer with task execution. We show that this abstraction can express a variety of shuffle algorithms, including dynamic strategies to handle data skew and stragglers (§3.3).

Although many distributed futures implementations exist, none of the these systems have been able to match the scale and performance of a monolithic shuffle system. CIEL [61] is the first to show MapReduce programs can be implemented using distributed futures, but it

lacks an in-memory object store which is crucial for efficient pipelining and data transfers. Dask [79], another distributed futures-based dataframe system, supports in-memory objects but cannot scale beyond hundreds of GBs (§3.5.4.1). Previous versions of Ray [59] support shuffle within the capacity of its distributed shared memory object store, but lack disk spilling mechanisms and therefore do not support out-of-core processing.

In this work, we extend Ray with the necessary features to support large-scale shuffle (§3.4). These include: (1) locality scheduling primitives to enable colocating tasks to better exploit shuffle data locality; (2) a full distributed memory hierarchy with disk spilling and recovery; (3) asynchronous object fetching to pipeline task execution with disk and network I/O. We present Exoshuffle, a flexible and scalable library for distributed shuffle built on top of Ray. We demonstrate the advantages of this extensible shuffle architecture by showing that (§3.5):

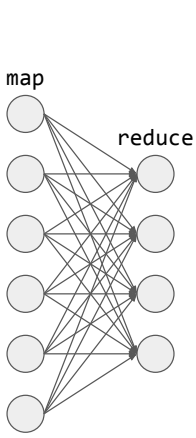
- A variety of previous shuffle optimizations can be written as distributed futures programs in Exoshuffle, with an order of magnitude less code.
- The Exoshuffle implementations of these shuffle optimizations match or exceed the performance of their monolithic counterparts.
- Exoshuffle can scale to 100 TB, outperforming Spark and Magnet by 1.8×, and breaking the CloudSort record as the world's most cost-efficient sorting system.
- Exoshuffle can easily integrate with a diverse set of applications such as distributed ML training, improving end-to-end training throughput by 2.4×.

3.2 Motivations

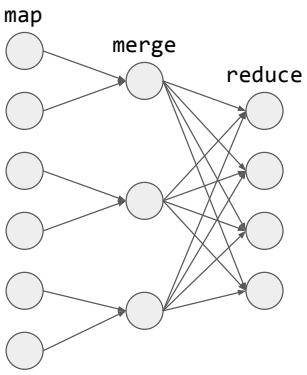
In this section, we overview two lines of previous work in building shuffle systems to illustrate the challenges in simultaneously achieving shuffle scalability and flexibility.

3.2.1 Shuffle Systems

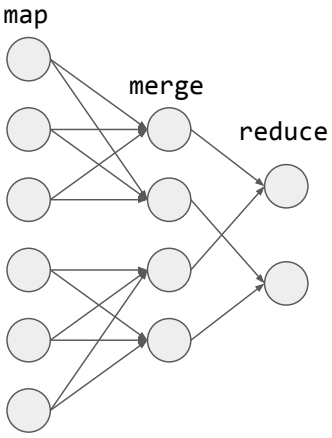
In a MapReduce operation with M map tasks and R reduce tasks, shuffle creates $M \times R$ intermediate blocks. Each of these blocks must be moved across memory, disk, and network. As the number of tasks grow, the number of blocks increases and the block size decreases both quadratically. At terabyte scale, this can result in hundreds of millions of very small blocks. This creates great challenges for I/O efficiency, especially for hard drives with low IOPS limits. Many shuffle systems have been built to optimize I/O efficiency in different storage environments. Table 3.1 shows an incomplete list of these systems, grouped by their target storage environments.



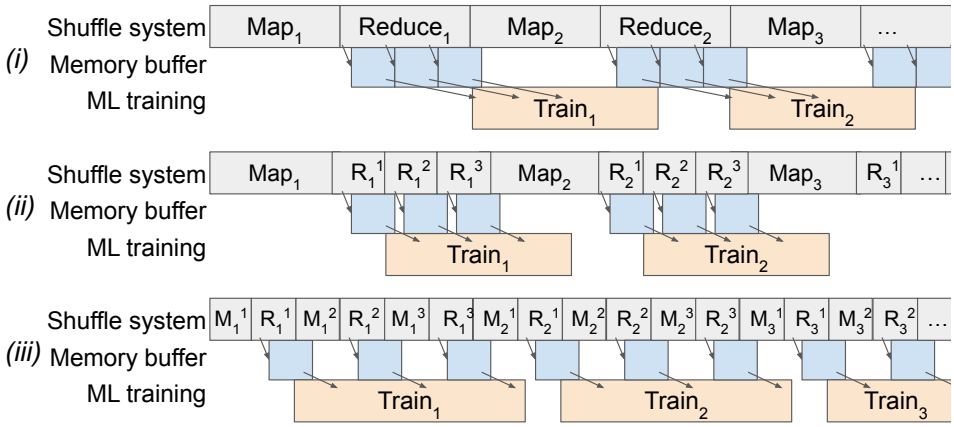
(a) "Simple" shuffle [26].



(b) Pre-shuffle merge [110].



(c) Push-based shuffle [84, 16, 95].



(d) ML training pipeline.

Figure 3.2: Shuffle algorithms for various applications. Exoshuffle uses distributed futures to execute these DAGs.

Previous I/O optimizations fall under two general categories: (1) reducing the number of small and random I/O accesses by *merging* intermediate blocks into larger ones at various stages [110, 16, 84] (Figures 3.2b and 3.2c), and (2) using *pipelining* to overlap I/O with execution [84, 39]. For example, *push-based shuffle* [95, 38] involves pushing intermediate outputs directly from the mappers to the reducers, allowing network and disk I/O to be overlapped with map execution, and optionally merging results on the reducer (Figure 3.2c) to improve disk write efficiency [84, 16].

While these solutions can improve throughput, they also come with high development cost. Each new operation, such as reduce-side merge, requires building additional protocols for managing block transfers. However, although the ideas may be system-agnostic,

Storage target	Shuffle systems
Hard disk	Sailfish [75], Riffle [110], Magnet [84]
SSD	Zeus [13]
Cloud storage	Alibaba E-MapReduce Shuffle [4], AWS Glue Shuffle [11], Google Cloud Dataflow Shuffle [85]

Table 3.1: Different shuffle systems are built to optimize shuffle for deployment in different storage environments.

the physical artifacts are often tightly integrated with proprietary storage systems, making them difficult to port to open-source frameworks. For example, many cloud providers build proprietary shuffle services to work with their own disaggregate storage offerings [11, 4, 85]; meanwhile, Magnet [84] is open-sourced as part of Spark but has yet to support disaggregated storage.

Furthermore, large-scale shuffle systems often come with more complicated deployment models. They are often deployed as auxiliary services to existing data processing systems. Shuffle services decouple block lifetimes from task executors to minimize interruptions upon executor failures [103], which are more frequent in large clusters. Shuffle services are also used to coordinate more sophisticated shuffle protocols, such as push-based shuffle and reduce-side merge [84]. However, because these shuffle services are only necessary at very large scale, they are not enabled by default in systems like Spark and require a separate deployment process.

Thus, while there has been significant innovation in new shuffle designs, few of these are widely deployed. Furthermore, it is difficult for an application to choose on the fly whether to use a particular shuffle algorithm; it requires both a priori knowledge of the application scenario and potentially an entirely different system deployment.

3.2.2 Random Shuffle in ML Training Pipelines

While much of the existing shuffle literature has focused on large-scale batch processing, there is also a need for performant shuffle in other application scenarios, such as online aggregation [24] and pipelining with more complex applications. An example of the latter is the *random shuffle* operation commonly used in machine learning training jobs. Note that by random shuffle, we mean the application-level transform that randomly permutes the rows of a dataset, rather than the generic system-level shuffle that is used to execute MapReduce applications.

To improve model convergence in deep learning, it is common practice to randomly shuffle the training dataset before feeding into GPU trainers to avoid bias on the order of

the data [57, 56]. To minimize GPU pauses, the shuffle should be pipelined with the training execution (Figure 3.2d). Furthermore, it is desirable for developers to be able to trade off between performance and accuracy: they might wish to run shuffle in a smaller window to reduce training latency, at the cost of overall end model accuracy.

These differences make it difficult for ML pipelines to directly leverage existing monolithic shuffle systems. Systems like Hadoop and Spark are highly optimized for global shuffle operations, but are not designed to pipeline the shuffle with downstream executions: shuffle results cannot be read until the full shuffle is complete [24]. The results must be written out to an external store before they can be read by the training workers (Figure 3.2d*i*). However, this leads to either high memory footprint, as it requires holding an additional copy of the dataset, or higher I/O overhead, if the shuffled data is written to disk before transfer to the GPU.

Fine-grained pipelining can improve efficiency. Figure 3.2d*ii* shows an example in which the reduce tasks for a particular epoch are pipelined with the training computation. This allows results to be used as they become available while limiting memory footprint to a single partition. Alternatively, the application can also choose to shuffle the dataset in windows (Figure 3.2d*iii*), improving pipelining at the cost of accuracy. Unfortunately, existing shuffle systems are not built for such fine-grained pipelining, and most big data systems that offer high-performance shuffle use an execution model that is incompatible with deep learning systems [25].

Instead, ML training frameworks often end up re-implementing shuffle within specialized data loaders and thus run into known problems that have been solved by traditional shuffle systems. Typically, data loaders are implemented with a pool of CPU-based workers colocated with the GPU trainers [62, 89, 37]. Each worker loads a partition of the dataset from storage (e.g., Amazon S3), preprocesseses it, and feeds the resulting data into the colocated trainers. To support random shuffle, the workers may read a random partition of the dataset on each epoch. However, to improve I/O efficiency, data must still be read in batches. Thus, to de-correlate data within the same batch, workers further shuffle the data by mixing records within a fixed-size local memory buffer. This effectively ties the shuffle window size to the size of the memory buffer. Setting the buffer size too large results in out-of-memory errors and poor pipelining, but if the buffer is too small, data de-correlation may be insufficient. In Section 3.5.3.3, we demonstrate how Exoshuffle can bring distributed shuffle optimizations to ML training applications, achieving both high performance and flexibility.

3.3 Shuffle with Distributed Futures

For distributed futures to serve as an intermediate abstraction layer for shuffle, they should: (1) abstract out the common implementation details of different shuffle implementations, (2) be general enough to allow heterogeneous end applications to interface with the shuffle

library, and (3) provide the same performance and reliability as monolithic shuffle systems. This narrow waist for distributed shuffle would enable both faster development for new shuffle implementations and extensibility to new application use cases.

Monolithic shuffle systems use messaging primitives, like RPC, as an intermediate abstraction layer. RPC is both general-purpose and high-performance, but it is too low-level to be a useful intermediate layer for shuffle. Integrating push-based shuffle into Spark, for example, required 1k+ LoC for the RPC layer changes alone [83]. Much of this development effort lies in implementing new inter-task protocols for data transfer and integrating them alongside existing ones.

In contrast, distributed futures decouple the shuffle control plane from the data plane. This abstraction enables different shuffle libraries to share a common data plane. Optimizations like push-based shuffle can be implemented in an order of magnitude less code as a result (§3.5.2).

In this section, we show how to express previous shuffle optimizations and application-specific shuffle variants as application-level programs with the distributed futures API. These simplified examples capture the logical execution DAG of the shuffle. Section 3.4 describes the physical execution of these programs and the details in achieving performance parity with monolithic shuffle systems.

3.3.1 The Distributed Futures API

A distributed futures program invokes remote functions, known as *tasks*, that execute and return data on a remote node. When calling a remote function, the caller immediately gets a distributed future that represents the *eventual* return value. The future is "distributed" because the return value may be stored anywhere in the cluster, e.g., at the node where the task executes. This avoids copying return values back to the caller, which can become expensive for large data.

The caller can make use of a distributed future in two ways. First, it can create a DAG by passing a distributed future as an argument to another task. The system ensures that the dependent task runs only after all of its arguments are computed. Note that the caller can specify such dependencies before the value is computed and that the caller need not see the physical values. This gives the system control over parallelism and data movement, e.g., pipelining task execution with dependency fetching for other tasks, and allows the caller to manipulate data larger than local memory. Second, the caller can get the value of a distributed future using a `get` call, which fetches the value to the caller's local memory. This is useful when consuming the output of a shuffle, as it allows the caller to pipeline its own execution with the shuffle. The caller can additionally use a `wait` call, which blocks until a set of tasks complete (without fetching the return values), for synchronization and for avoiding scheduling too many concurrent tasks.

3.3.2 Expressing Shuffle with Distributed Futures

We demonstrate how these APIs can be used to express various shuffle optimizations (Fig 3.2) as application-level programs. We use Ray's distributed futures API for Python [59] for illustration. The `@ray.remote` annotation designates remote functions, and the `.remote()` operator invokes tasks.

Listing 2 Shuffle algorithms as distributed futures programs.

```
def simple_shuffle(M, R, map, reduce):
  map_out = [map.remote(m) for m in range(M)]
  return ray.get([
    reduce.remote(map_out[:,r]) for r in range(R)]

def shuffle_riffle(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
    merge.remote(map_out[i*F:(i+1)*F, :])
    for i in range(M/F)]
  return ray.get([
    reduce.remote(merge_out[:,r]) for r in range(R)]

def shuffle_magnet(M, R, F, map, reduce, merge):
  map_out = [map.remote(m) for m in range(M)]
  merge_out = [
      [merge.remote(map_out[i*F:(i+1)*F, r])
      for i in range(M/F)] for r in range(R)]
  return ray.get(
    [reduce.remote(merge_out[:,r]) for r in range(R)]
```

3.3.2.1 Simple Shuffle

In Listing 2, `simple_shuffle` shows a straightforward implementation of the MapReduce paradigm illustrated in Figure 3.2a. The shuffle routine takes a `map` function that returns a list of map outputs, and a `reduce` function that takes a list of map outputs and returns a reduced value. `M` and `R` are the numbers of map and reduce tasks respectively. The two statements produce the task graph shown in Figure 3.2a. Note that the `.remote()` calls are non-blocking, so the entire task graph can be submitted to the system without waiting for any one task to complete.

This is effectively *pull-based shuffle*, in which shuffle blocks are *pulled* from the map workers as reduce tasks progress. Assuming a fixed partition size, the total number of shuffle blocks grows quadratically with the total data size. Section 3.5.1 shows empirical evidence

of this problem: as the number of shuffle blocks increases, the performance of the naive shuffle implementation drops due to decreased I/O efficiency. Prior work [110, 84, 16] have proposed solutions to this problem, which we study and compare next.

3.3.2.2 Pre-Shuffle Merge

Riffle [110] is a specialized shuffle system built for Spark. Its key optimization is merging small map output blocks into larger blocks, thereby converting small, random disk I/O into large, sequential I/O before shuffling over the network to the reducers. The merging factor F is either pre-configured, or dynamically decided based on a block size threshold. As soon as F map tasks finish on an executor node, their output blocks ($F \times R$) are merged into R blocks, each consisting of F blocks of data from the map tasks. This strategy, illustrated in Figure 3.2b, is implemented in Listing 2 (`shuffle_riffle`). The code additionally takes `F` as the merging factor, and a merge function which combines multiple map outputs into one.

Riffle's key design choice is to merge map blocks *locally* before they are pulled by the reducers, as shown in the highlighted lines. For simplicity, the code assumes that the first F map tasks are scheduled on the first worker, the next F map tasks on the second worker, etc. In reality, the locality can be determined using scheduling placement hints or runtime introspection (§3.4.2) Section 3.5.1 shows that this implementation of Riffle-style shuffle improves the job completion time over simple shuffle.

3.3.2.3 Push-based Shuffle

Push-based shuffle (Fig 3.2c) is an optimization that pushes shuffle blocks to reducer nodes as soon as they are computed, rather than pulling blocks to the reducer when they are required. Magnet [84] is a specialized shuffle service for Spark that performs this optimization by merging intermediate blocks on the reducer node before the final reduce stage. This improves I/O efficiency and data locality for the final reduce tasks. `shuffle_magnet` in Listing 2 implements this design.

3.3.2.4 Straggler Mitigation

Distributed futures enable dynamic task graphs by nature, making it ideal for detecting and reacting to stragglers during runtime.

Speculative Execution One way to handle stragglers is through speculative execution. Tasks that are suspected to be stragglers can be duplicated, and the system chooses whichever result is available first. This can be accomplished with distributed futures using the `ray.wait` primitive, as shown in Listing 3.

Listing 3 Mitigating stragglers with speculative execution.

```
map_out = ...
_, timeout_tasks = ray.wait(map_out, timeout=TIMEOUT)
duplicates = []
for task in timeout_tasks:
  duplicates.append(map.remote(task.args))
for t1, t2 in zip(timeout_tasks, duplicates):
  t, _ = ray.wait([t1, t2], num_returns=1)
  map_out[t1.id] = t
```

Best-effort Merge Shuffle systems including Riffle and Magnet also implement "best-effort merge", where a timeout can be set on the shuffle and merge phase [110, 84]. If some merge tasks are cancelled due to timeout, the original map output blocks will be fetched instead. This ensures straggler merge tasks will not block the progress of the entire system. Best-effort merge can be implemented in Exoshuffle as shown in Listing 4 using an additional `ray.cancel()` API which cancels the execution of a task. The cancelled task's input, which are the original map output blocks, will then be directly passed to the reducers. This way, the task graph is dynamically constructed as the program runs, adapting to runtime conditions while still enjoying the benefits of transparent fault tolerance provided by the system.

Listing 4 Mitigating stragglers via task cancellation.

```
map_out = ...
merge_out = ...
_, timeout_tasks = ray.wait(merge_out, timeout=TIMEOUT)
for task in timeout_tasks:
  ray.cancel(task)
  merge_out[task.id] = task.args
out = [reduce.remote(merge_out[:, r]) for r in range(R)]
ray.wait(out)
```

3.3.2.5 Data Skew

Data skew can be prevented at the data management level using techniques such as key salting, or periodic repartitioning. However, it is still possible for skews to occur during ad-hoc query processing, especially for those queries involving joins and group-bys. Data skew during runtime can cause the working set of a reduce task to be too large to fit into executor memory.

Dynamic repartitioning solves this problem by further partitioning a large reducer partition into smaller ones. This is straightforward to implement since the distributed futures

programming model enables dynamic tasks by nature. Listing 5 shows that we can recursively split down a reducer's working set until it fits into a predefined memory threshold.

Listing 5 Dynamic repartitioning for skewed partitions.

```
@ray.remote
def reducer(*parts):
  total_size = [part.size() for part in parts]
  if total_size > THRESHOLD:
    L = len(parts) // 2
    return flatten([
      reducer.remote(*parts[:L]),
      reducer.remote(*parts[L:])])
```

3.3.3 Applications

Because Exoshuffle implements shuffle at the application level, it can easily interoperate with other applications. Here, we demonstrate two example applications that use fine-grained pipelining with shuffle to improve end-to-end performance. These applications are evaluated in Section 3.5.3.

3.3.3.1 Online Aggregation with Streaming Shuffle

Online aggregation [42] is an interactive query processing mode where partial results are returned to the user as soon as some data is processed, and are refined as progress continues. This is especially useful when the query takes a long time to complete. Online aggregation is difficult to implement in MapReduce systems because they require all outputs to be materialized before being consumed. Past work made in-depth modifications to Hadoop and Spark to support online aggregation [24, 109].

Online aggregation is straightforward to implement in Exoshuffle without the need to modify the underlying distributed futures system. Listing 6 shows the `streaming_shuffle` routine. It requires a modified `reduce` function that takes a reducer state and a list of map outputs and returns an updated state, and an `aggregate` function which combines the reducer states to produce aggregate statistics. Shuffle is executed in rounds. At the end of each round, the aggregation function is invoked with the reducer outputs, and will asynchronously print an aggregate statistic (e.g. sum) to the user. Note that the Exoshuffle user can simply swap between `simple_shuffle` and `streaming_shuffle` to get the semantics they desire.

Listing 6 Streaming shuffle and pipelined data loading for ML.

```
1  def streaming_shuffle(map, reduce, print_aggregate):
2    reduce_states = [None] * R
3    for rnd in range(N):
4      map_results = [map.remote(M*rnd+i) for i in range(M)]
5      ray.wait(reduce_states)
6      reduce_states = [
7        reduce.remote(reduce_state, *map_results[:, r])
8        for r, reduce_state in enumerate(reduce_states)]
9      print_aggregate.remote(reduce_states)
10   return ray.get(reduce_states)
11
12 def model_training(trainer, data):
13   shuffle_out = shuffle(data, ...)
14   for epoch in range(EPOCHS):
15     next_shuffle_out = shuffle(data, ...)
16     for block in shuffle_out:
17       trainer.train(ray.get(block))
18     shuffle_out = next_shuffle_out
```

3.3.3.2 Distributed ML Training with Pipelined Shuffle

Exoshuffle also enables fine-grained pipelining for ML training, as illustrated in Figure 3.2d. In Listing 6, `model_training` shows the code skeleton. On line 13, the `shuffle` function (could be any in Listing 2) returns a set of distributed futures pointing to reducer outputs. They are passed immediately to the model trainer while shuffle executes asynchronously. As soon as a reducer block becomes available, the model trainer acquires it (line 17) and send it to the GPU for training. This achieves the fine-grained pipelining described in Figure 3.2d.

3.4 System Architecture

Section 3.3 shows how shuffle DAGs can be expressed as distributed futures programs. However, achieving high performance shuffle also requires a set of critical system facilities. In this section, we describe the architecture of Exoshuffle via a realistic implementation of the push-based shuffle described in Section 3.3.2.3. We describe the additional system APIs used by Exoshuffle (§3.4.2), and the transparent features provided by the underlying distributed futures implementation (§3.4.3) that are key to performance.

3.4.1 Example: Push-based Shuffle

Listing 7 Implementation of two-stage shuffle.

```
1  def push_based_shuffle(map, reduce):
2    @ray.remote
3    def merge(*map_results):
4      for results in zip(*map_results):
5        yield reduce(*results)
6
7    merge_results = numpy.empty((NUM_WORKERS,
8      NUM_ROUNDS, NUM_REDUCERS_PER_WORKER))
9
10   # Map and shuffle.
11   for rnd in range(NUM_ROUNDS):
12     for i in range(NUM_TASKS_PER_ROUND):
13       map_results = [
14         map.options(num_returns=NUM_WORKERS).remote(
15           parts[rnd * NUM_TASKS_PER_ROUND + i])
16         for i in range(NUM_TASKS_PER_ROUND)]
17
18     if rnd > 0:
19       ray.wait(merge_results[:, rnd - 1, :])
20
21     for w in range(NUM_WORKERS):
22       merge_results[w, rnd, :] = merge.options(
23         worker=w, num_returns=NUM_REDUCERS_PER_WORKER
24       ).remote(*map_results[:, w])
25     del map_results
26
27   # Reduce.
28   return flatten(
29     [[reduce.remote(*merge_results[w, :, rnd])
30       for rnd in range(NUM_REDUCERS_PER_WORKER)]
31       for w in range(NUM_WORKERS)])
```

Listing 7 implements push-based shuffle (§3.3.2.3) for a cluster of `NUM_WORKERS` nodes. The library takes a `map` and a `reduce` function as input. The remaining constants are chosen by the library according to the user-specified number of input and output partitions.

Lines 11–25 comprise the map and merge stage, in which map results are shuffled, pushed to the reducer nodes, and merged. This stage pipelines between CPU (`map` and `merge` tasks), network (to move data between `map` and `merge`), and disk (to write out `merge` results). The map and merge tasks are scheduled in rounds for pipelining: Lines 18–19 ensures that there is at most one round of merge tasks executing, and that they can overlap with the following round's map tasks. Each round submits one merge task per worker node. Each merge task

takes in one intermediate result from each map task from the same round and returns as many merged results as there are reduce partitions on that worker.

Once all map and merge tasks are complete, we schedule all reduce tasks (lines 28–31) and return the distributed future results. Each reduce task performs a final reduce on all merge results for its given partition. To minimize unnecessary data transfer, the reduce tasks are co-located with the merge tasks whose results they read.

3.4.2 Scheduling Primitives

For complex applications like distributed shuffle, it is difficult for a general-purpose system to make optimal decisions in every context. For instance, optimally scheduling a computation DAG on a set of nodes is NP-hard [19]. It is therefore more robust to allow the application or library developer to apply domain-specific knowledge to achieve better performance.

By default, Ray provides a two-level distributed scheduler that balances between bin-packing vs. load-balancing [59]. This is sufficient for map and reduce tasks in simple shuffle, as these can be executed anywhere in the cluster. However, more advanced shuffle strategies (§§3.3.2.2 and 3.3.2.3) require more careful placement and scheduling of tasks to improve performance. In this section, we describe the additional APIs designed to give the shuffle library more control over the physical execution of the shuffle DAG.

3.4.2.1 Scheduling for Data Locality

Ray provides automatic locality-based scheduling when possible. For example in Listing 7, lines 28–31, Ray automatically schedules the reduce tasks on the workers on which the upstream merge results reside. In some other cases, hints must be provided to the system to achieve better data locality. For example, a group of merge tasks must be colocated with the downstream reduce task, but this is impossible for the system to determine because the reduce task's dependency is not known to the system yet. To handle this problem, we introduce *node-affinity scheduling* in Ray, which allows the application to pin tasks to a particular node. For example, Listing 7 uses this in line 23 to colocate merge tasks for the same reducer. Node affinity is soft, meaning that Ray will choose another suitable node if the specified node fails.

3.4.2.2 Scheduling for Task Pipelining

The map and merge tasks should be pipelined to allow map results to be shuffled concurrently with map execution. This task-level pipelining is challenging for a distributed futures system to determine automatically: Too many concurrent map tasks will reduce resources available to downstream merge tasks, and scheduling the wrong set of map and merge tasks

concurrently prevents map outputs from being consumed directly by merge tasks, resulting in unnecessary disk writes. The shuffle library is better placed to determine that it should apply backpressure by limiting the number of concurrent map and merge tasks. The library can also determine that a round of merge tasks should be executed concurrently with the following round of map tasks. Exoshuffle achieves this with the `wait` API (Listing 7, line 19), which blocks until a task completes.

3.4.2.3 Controlling Redundancy with Reference Counting

Distributed futures are reference-counted in Ray. While an object reference is in scope, Ray attempts to ensure its value exists in the cluster. By selecting which references to keep or drop, the shuffle library can make tradeoffs between reducing write amplification and improving data redundancy. For example, line 25 of Listing 7 deletes the intermediate map results from the current round. This reduces write amplification, as the map results can be immediately dropped from memory without spilling to disk, but requires additional re-execution upon failure. Alternatively, the shuffle library can instead keep the intermediate references, resulting in additional disk writes but improved data redundancy.

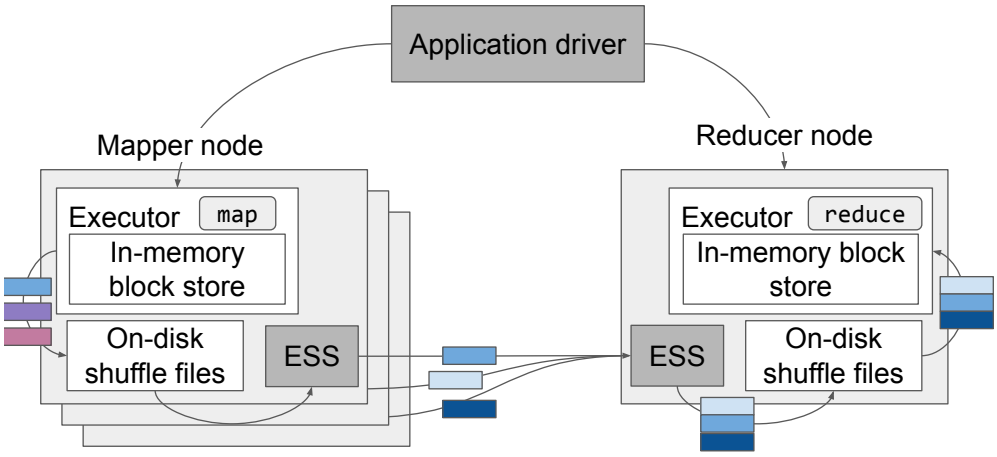
3.4.3 Transparent System Facilities

The actual data transfer, or *shuffle*, is managed by the distributed futures system according to the application specifications. For example in Listing 7, lines 23–25 specifies that one column of the distributed futures in `map_results` should be sent to one merge task. This prompts the data plane to transfer the corresponding physical data to the `merge` task's location. In this section, we describe the transparent storage and I/O mechanisms provided by the distributed futures system to facilitate this data movement.

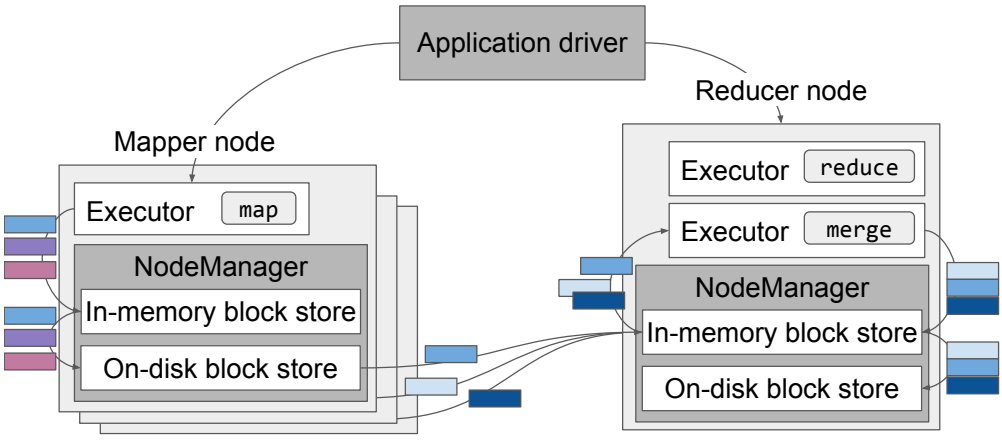
3.4.3.1 Shared Memory Object Store

Previous monolithic shuffle systems implement distributed coordination via an *external shuffle service*, a specialized process deployed to each node that orchestrates block transfers (Fig 3.3a). This process is external to the executors, decoupling block transfers from map and reduce task execution. In Exoshuffle, we replace this service with a generic node manager that is responsible for both in-memory and spilled objects (Fig 3.3b).

We build on Ray's shared memory object store [59] for immutable objects. Each node manager hosts a shared memory object store shared by all executors on that node (Fig 3.3b). This decouples executors from blocks: once a task's outputs are stored in its local object store, the node manager manages the block. This keeps executors stateless and allows them to execute other tasks or exit safely while the node manager coordinates block movement. Shared memory enables *zero-copy* reads of object data on the same node, which avoids CPU



(a) Example of a monolithic shuffle architecture.



(b) Exoshuffle.

Figure 3.3: Comparing a monolithic vs. application-level shuffle architecture. (a) implements all coordination and block management through an external shuffle service on each node, in this case implementing the Magnet shuffle strategy (§3.3.2.3). (b) shows the same shuffle strategy but implemented as an application on a generic distributed futures system.

and memory overhead. By making objects immutable, we also avoid consistency concerns between object copies.

Next, we describe extensions to the original Ray architecture [59] made in this work that improves pipelining disk and network I/O with task execution. These improvements are made at the system level without knowledge of the application-level shuffle semantics, and thus can benefit a wide range of data-intensive applications.

3.4.3.2 Pipelined Object I/O

Object Allocation and Fetching. There are two categories of object memory allocations: new objects created for task returns (e.g., `map` task outputs), and copies of objects fetched remotely as task arguments (e.g., `merge` task inputs). The memory subsystem queues and prioritizes object allocations to ensure forward progress while keeping memory usage bounded to a limit. This is critical for reducing thrashing within the object store, caused by requesting objects for too many concurrent requests, while leaving sufficient heap memory for task executors.

All memory allocations on a Ray worker node go into an allocation queue for fulfillment. If there is spare memory, the allocation is fulfilled immediately. Otherwise, requests are queued until the spilling process or garbage collection frees up enough memory. If memory is still insufficient, Ray falls back to allocating task output objects on the filesystem to ensure liveness. Spare memory besides the memory allocated to executing task arguments and returns is used to fetch the arguments of queued tasks. This enables pipelining between execution and I/O, i.e. restoring objects from disk or fetching objects over the network. For example, at line 28 in Listing 7, all merge results are already spilled to disk and all reduce tasks are submitted at once. While earlier reduce tasks execute, the system uses any spare memory to restore merge results for the next round of reduce tasks from disk.

Object Spilling. Object spilling is transparent, so the application need not specify if or when it should occur. When the memory allocation subsystem has backlogged requests, the spilling subsystem migrates referenced objects to disk to free up memory. When a spilled object's data is required locally for a task, e.g., because it is the argument of a queued task, the node manager copies it back to memory as described above. When requested by a remote node, the spilled object is streamed directly from disk across the network to the remote node manager. To improve I/O efficiency, Ray coalesces small objects into larger files before writing to the filesystem.

3.4.3.3 Fault Tolerance

Exoshuffle relies on lineage reconstruction for distributed futures to recover objects lost to node failures [99], a similar mechanism to previous shuffle systems [26, 107]. In Ray, the application driver stores the object lineage and resubmits tasks as needed upon failure. This process is transparent to Exoshuffle, which runs at the application level. Still, Exoshuffle can use object references (§3.4.2.3) to specify reconstruction or eviction for specific objects.

Executor process failures are much more common than node failures. If reconstruction is required each time an executor fails, it can impede progress [103, 84]. Many previous shuffle systems use an external shuffle service to ensure map output availability in the case of executor failures or garbage collection pauses. Similarly, in Exoshuffle, executor process

failures do not result in the loss of objects, because the object store is run inside the node manager as a separate process.

More sophisticated shuffle systems require additional protocols such as deduplication to ensure fault tolerance [16]. Distributed futures prevent such inconsistencies because they require objects to be immutable, task dependencies to be fixed, and tasks to be idempotent.

To reduce the chance of data loss, some shuffle system uses on-disk [84] or in-memory [16] replication of intermediate blocks to guard against single node failures. In Ray, objects are spilled to disk and transferred to remote nodes where they are needed, which also results in multiple copies as long as the object is in scope. The application can also disable this optimization by deleting its references to the object (Listing 7, L25). In the future, we could allow the application to more finely tune the number of replicas kept, e.g., by passing this as a parameter during task invocation.

3.5 Evaluation

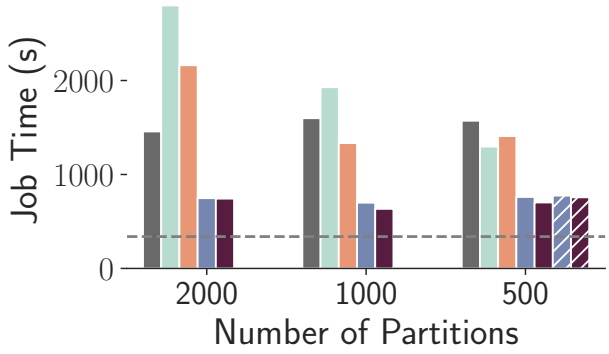
We study the following questions in the evaluation:

- Can Exoshuffle libraries achieve performance and scalability competitive with monolithic shuffle systems? (§3.5.1)
- Is it easier to implement shuffle optimizations in Exoshuffle? (§3.5.2)
- What benefits does Exoshuffle provide for applications, including CloudSort, online aggregation and ML training? (§3.5.3)
- How do the features in the distributed futures backend contribute to Exoshuffle performance? (§3.5.4)

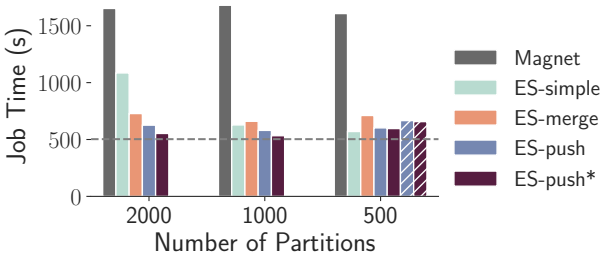
3.5.1 Shuffle Performance

3.5.1.1 Setup

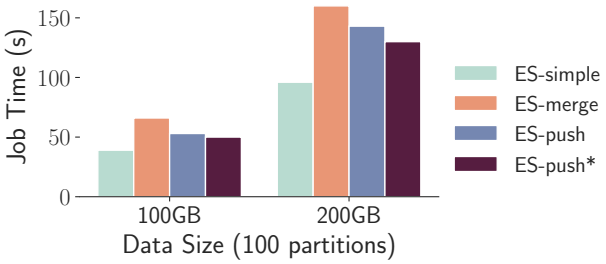
We create test environments on Amazon EC2 using VMs targeted at data warehouse use cases. We test on a HDD cluster of d3.2xlarge instances (8 CPU, 64 GiB RAM, 6× HDD, 1.1 GB/s aggregate sequential throughput, 18K aggregate IOPS, 15 Gbps network), and a SSD cluster of i3.2xlarge instances (8 CPU, 61 GiB RAM, NVMe SSD, 720 MB/s throughput, 180K write IOPS, 10 Gbps network).



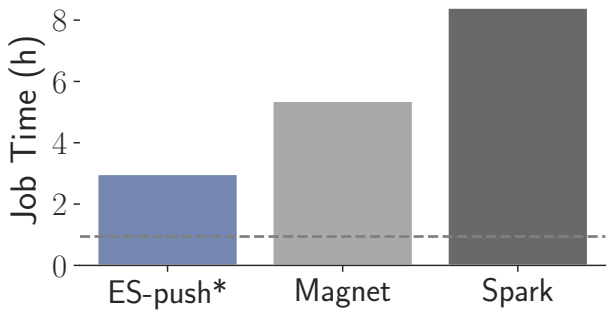
(a) 1 TB sort on 10 HDD nodes.



(b) 1 TB sort on 10 SSD nodes. Semi-shaded bars are runs with failures (§3.5.4.3).



(c) In-memory sort on 10 SSD nodes.



(d) 100 TB on 100 HDD nodes.

Figure 3.4: Comparing job completion times on the Sort Benchmark. The dashed lines indicate the theoretical baseline (§3.5.1.1). Exoshuffle is abbreviated as ES.

Workload. We run the Sort Benchmark (a.k.a. TeraSort or CloudSort) [82], as it is a common benchmark for testing raw shuffle system performance. This benchmark requires sorting a synthetic dataset of configurable size, consisting of 100-byte records with 10-byte keys.

Baselines. We compare to the push-based shuffle service in Spark, a.k.a. Magnet, and a theoretical baseline.

Magnet is integrated into Spark in its 3.2.0 release as an external push-based shuffle service. We run Spark 3.2.0 on Hadoop 3.3.1 with Magnet shuffle service enabled. We disable compression of shuffle files according to the rules of TeraSort. This allows for a fair comparison in terms of total bytes of disk I/O.

For the theoretical baseline, we assume disk I/O is the bottleneck since empirically we find that disk I/O takes longer than networking and CPU processing in this benchmark. The baseline is calculated by $T = 4D/B$, where D is the total data size and B is the aggregate disk bandwidth. D is multiplied by 4 because each datum needs to be read twice and written twice, a theoretical minimum for external sort [76].

Exoshuffle variants. We run Exoshuffle on Ray 1.11.0. We compare implementations of the following shuffle libraries:

- ES-simple, the simple shuffle variant (§3.3.2.1).
- ES-merge, pull-based shuffle with pre-shuffle merge, similar to that in Riffle (§3.3.2.2).
- ES-push, push-based shuffle similar to Magnet (§3.3.2.3).
- ES-push*, push-based shuffle further optimized to reduce write amplification (§3.4.1).

3.5.1.2 Performance Comparison of Shuffle Algorithms

Performance on HDD. Figure 3.4a shows the job completion times of Exoshuffle variants running 1 TB sort on 10 HDD nodes. ES-simple shows the well-known scaling problem: performance degrades as the number of partitions increases, because the intermediate shuffle blocks become more in number and smaller in size both quadratically, quickly reaching disk IOPS limit. The push-based shuffle variants (ES-push, -push*) achieve better performance regardless of the number of partitions, thanks to the merging of shuffle blocks to increase disk I/O efficiency and the pipelining of disk and network I/O. ES-merge runs slower than -simple because merging the map output blocks incurs additional disk writes, which outweighs the I/O efficiency savings when the number of partitions is small, and only shows benefits when the number of partitions increases. The Magnet baseline shows comparable performance. In summary, Exoshuffle libraries demonstrate performance benefits that match the characteristics of their monolithic counterparts.

Performance on SSD. Figure 3.4b shows the same benchmark and variants running on the SSD cluster. All variants of Exoshuffle outperform the PBS baseline, and display similar trends as on the HDD cluster. The run times of the optimized versions of Exoshuffle are also close to the theoretical baseline. Since the NVMe SSD supports much higher random IOPS, the I/O efficiency gains are less pronounced.

In-memory Performance. Figure 3.4c shows that when data fits in memory, ES-simple is actually the fastest algorithm compared to all other variants. This is because the other algorithms create copies of data by merging them, triggering unnecessary disk spilling. Magnet observes similar behavior for small datasets*.

Conclusion. These experiments show that the shuffle algorithms provided by Exoshuffle offer the same performance benefits as their monolithic counterparts. Furthermore, the most performant shuffle algorithm depends on the data size and hardware configuration, and Exoshuffle offers the flexibility to choose the most suitable algorithm at the application level, without having to deploy multiple systems.

3.5.1.3 Shuffle Scalability

To test performance at large scale, we run the Sort Benchmark on 100 TB data with 50 000× 2 GB input partitions on a cluster of 100× d3.2xlarge VMs. For Exoshuffle, we run the ES-push* variant since it is the most optimized for scale. For baselines, we run both Spark's native shuffle (Spark) and its push-based shuffle service (Magnet). We run both baselines with compression on because Spark without compression becomes unstable at this scale.

Figure 3.4d shows the results. Exoshuffle outperforms both native Spark shuffle and the push-based shuffle service Magnet, despite Spark's compression reducing total bytes spilled by 40%. Magnet improves shuffle performance by 1.6× because it reduces random disk I/O. Exoshuffle further improves performance over push-based Spark by 1.8×. This difference comes from reduced write amplification in ES-push*, which spills only the merged map outputs, while Magnet also spills the un-merged map outputs. These additional writes provide faster failure recovery through improved durability, albeit at the cost of performance. Exoshuffle allows the application to choose between these tradeoffs by using ES-push vs. ES-push*.

*The Spark 3.3.1 documentation states: "Currently [Magnet] is not well suited for jobs/queries which runs quickly dealing with lesser amount of shuffle data."

†Total lines of code in `org.apache.spark.shuffle`.

‡As reported by Zhang et al. [110]

§Total added lines in `https://github.com/apache/spark/pull/29808/files`.

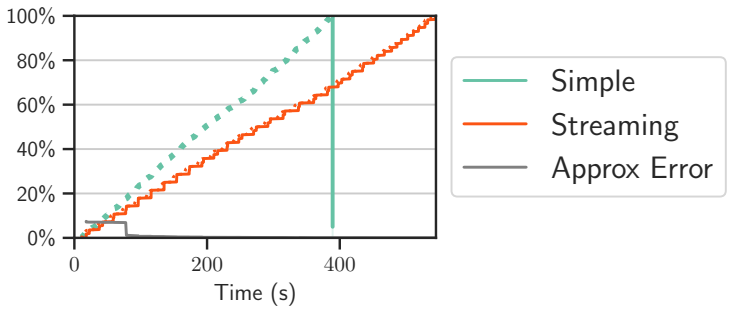


Figure 3.5: Online aggregation. Dotted lines show map progress; solid lines show reduce progress.

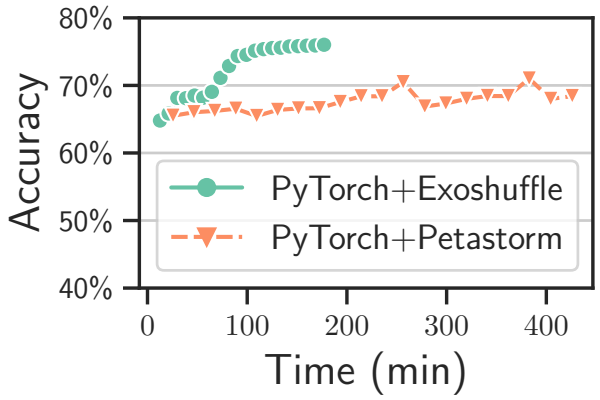


Figure 3.6: Single-node ML training for 20 epochs.

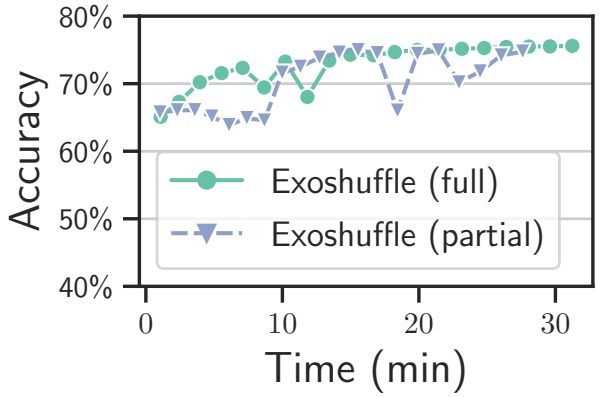


Figure 3.7: 4-node, distributed ML training for 20 epochs.

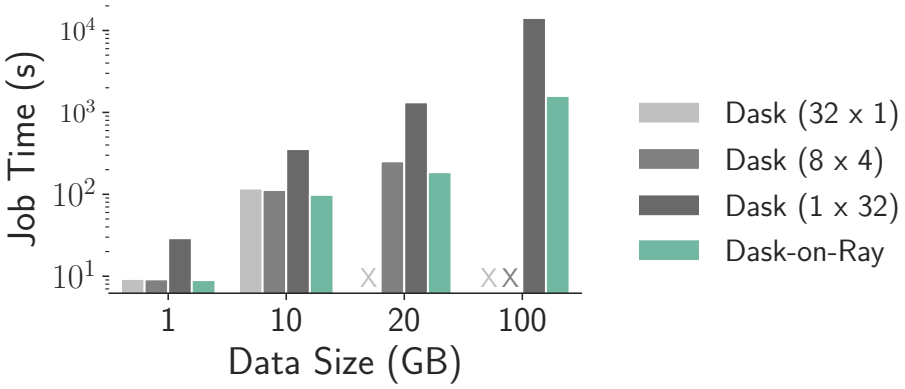


Figure 3.8: Comparing shuffle time in Dask and Ray. Legends show number of processes × threads.

Figure 3.9: Effect of I/O optimizations in Ray.

Shuffle Algorithm	System LoC	Exoshuffle LoC
Simple (§3.3.2.1)	2600 (Spark†)	**215**
Pre-shuffle merge (§3.3.2.2)	4000 (Riffle‡)	**265**
Push-based shuffle (§3.3.2.3)	6700 (Magnet§)	**256**
with pipelining (§3.4.1)	–	**256**

Table 3.2: Approximate lines of code for implementing shuffle algorithms in Exoshuffle versus in specialized shuffle systems.

3.5.2 Implementation Complexity

In Exoshuffle, shuffle is expressed as application-level programs. Table 3.2 compares the amount of code of several monolithic shuffle systems with the lines of code needed to implement the corresponding shuffle algorithms in Exoshuffle. Exoshuffle libraries may not provide all the production features of the monolithic counterparts, but many shuffle optimizations

can be implemented in Exoshuffle with an order of magnitude less code, while keeping the same performance benefits. By offering shuffle as a library, Exoshuffle also allows applications to choose the best shuffle implementation at run time without deploying multiple systems.

3.5.3 Shuffle Applications

Next, we show how Exoshuffle can extend distributed shuffle support for a broader set of applications.

3.5.3.1 CloudSort

The CloudSort competition [82] calls for the most cost-efficient way to sort 100 TB of data on the public cloud. We ran Exoshuffle-CloudSort on a cluster of 40× i4i.4xlarge nodes with input and output data stored on Amazon S3, and set a new world record of $0.97/TB [50]. This is 33% more cost-efficient than the previous world record set in 2016. The previous entry used a heavily modified version of Spark for the CloudSort workload [95]. In contrast, Exoshuffle-CloudSort is only hundreds of lines of application code running on a release version of Ray.

To account for the fact that the cloud hardware costs have lowered since 2016, we take the setup from the previous record-winning entry and look up its cost on today's Alibaba Cloud. Table 3.3 shows that the same amount of cloud resources would cost $115 today. Still, Exoshuffle-CloudSort achieves another 15% cost reduction beyond this result. We calculate another theoretical baseline of simply shuffling 100 TB data through the AWS network (without sorting), which would cost $74. This puts our record within 31% of the theoretical limit. This result demonstrates that Exoshuffle can achieve state-of-the-art performance and cost-efficiency for large-scale shuffle.

System	Cost
NADSort (2016)	$1.44/TB
NADSort (2022, extrapolated)	$1.15/TB
Exoshuffle-CloudSort (2022)	**$0.97/TB**

Table 3.3: CloudSort costs over years.

3.5.3.2 Online Aggregation with Streaming Shuffle

We use a 1 TB dataset containing 6 months of hourly page view statistics on Wikipedia. We run an aggregation to get the ranking of the top pages by language on 10× r6i.2xlarge nodes

with data loaded from S3. Figure 3.5 shows the difference between regular and streaming shuffle. The streaming shuffle takes 1.4× longer to run in total due to the extra computation needed to produce partial results. However, with streaming shuffle, the user can get partial aggregation results within 8% error[¶] of the final result in 18 seconds, 22× faster than regular shuffle. Exoshuffle makes it easy to switch between `simple_shuffle` and `streaming_shuffle` to choose between partial result latency and total query run time.

3.5.3.3 Distributed ML Training

Many distributed training frameworks already run on top of Ray. By offering Exoshuffle as a library, we enable these workloads to leverage scalable shuffle. We demonstrate Exoshuffle's ability to support fine-grained pipelining for ML training using the Ludwig framework [58] to train a deep classification model TabNet on the HIGGS dataset (7.5 GB). Ludwig integrates ML data loaders with the PyTorch training framework [68]. Efficient training requires randomly shuffling the data per epoch before sending it into the GPU for training.

We first run the ML training on a single g4dn.4xlarge instance. We compare two versions of Ludwig: Ludwig 0.4.0 uses Petastorm [37], which prefetches data in batches into a per-process memory buffer and performs random shuffle in the buffer. This approach makes the shuffle window size limited by the memory buffer size (§3.2.2).

In this experiment, we set the shuffle window size to 9% of the total data size to avoid OOM errors. In comparison, Ludwig 0.4.1 uses Exoshuffle offered through Ray Data [90]. It pipelines data loading and shuffling with GPU training (Fig 3.2d), and supports full shuffle across loading batches by storing data in the shared-memory object store. Figure 3.6 shows that model training with Exoshuffle is 2.4× faster end-to-end thanks to the fine-grained pipelining. The model also converges faster per-epoch and to a higher accuracy, because Exoshuffle performs complete random shuffling between epochs, whereas Petastorm's random shuffle is limited to subsets of the data.

Next, we run the training on 4 g4dn.xlarge nodes to show the distributed shuffle performance. Ludwig 0.4.x has known bugs with distributed training, so we could not compare Petastorm with Exoshuffle. Instead, we use the latest Ludwig 0.6.0 and compares two shuffle strategies with the Exoshuffle-based data loader: full shuffle (the default) and partial shuffle. For partial shuffle, we emulate the Petastorm behavior and perform random shuffling only in each in-memory batch. Figure 3.7 shows that per-epoch time is slightly faster with partial shuffle, since it is fully local, but the convergence accuracy is slightly lower because of the less random shuffling of training data. This example demonstrates that Exoshuffle gives the developer the flexibility to choose the best shuffle strategy based on their training needs, while providing high-throughput data loading and shuffling.

[¶]Error is computed using the KL-divergence $D_{KL} = \sum p \log(p/\hat{p})$ where p is the true statistic and $\hat{p}$ is the sample statistic.

3.5.4 System Microbenchmarks

The Exoshuffle architecture requires high-performance components from the distributed futures system to deliver good performance. In this section, we study the the impact of these system components on shuffle performance.

3.5.4.1 Shared-Memory Object Store

We study the effect of a shared-memory object store that decouples objects from executors by comparing Dask and Ray. Dask and Ray are both distributed futures systems, but they differ in architecture. Ray uses a shared-memory object store that is shared by multiple executor processes on the same node (§3.4.3.1). Dask stores objects in executor memory and requires the user to choose between multiprocessing and multithreading. With multithreading, multiple Dask executor threads share data in a heap-memory object store, but the Python Global Interpreter Lock can severely limit parallelism. Dask in multiprocessing mode avoids this issue but uses one object store per worker process, so objects must be copied between workers on the same node. Thus, the lack of a shared-memory object store results in either reduced parallelism (multithreading) or high overhead for sharing objects (multiprocessing). It is also less robust as objects are vulnerable to executor failures.

We study these differences by running the same Dask task graph on Dask and Ray backends [96]. Figure 3.8 shows dataframe sorting performance on a single node (32 CPU, 244 GB RAM, 100 partitions). For Dask, we vary the number of executor processes and threads to show the tradeoff between memory usage and parallelism. Ray requires no configuration and uses 32 executor processes, 1 per CPU.

On small data sizes, Dask with multiprocessing achieves about the same performance as Ray, but it is 3× slower with multithreading due to reduced parallelism. On larger data sizes, Dask with multiprocessing fails due to high memory pressure from extra object copies. Meanwhile, Ray's shared-memory object store enables better stability and lower run time on all data sizes.

3.5.4.2 Small I/O Mitigations

Ray implements two system-level optimizations for mitigating the small I/O problem: fusing writes of spilled objects to avoid small disk I/O, and prefetching task arguments to hide network and disk latency (§3.4.3.2). To show the impact of these optimizations, we run a single-node microbenchmark that creates 16 GB total objects in a 1 GB object store, forces them to spill to disk, then restores the objects from disk. We use object sizes ranging from 100 KB to 1 MB, as these are comparable to the shuffle block sizes. We use a `sc1` HDD disk since the disk I/O bottleneck is more pronounced on slower storage.

Fusing Writes. Ray fuses objects into at least 100 MB files then writes them to disk. Figure 3.9 shows the total run time stays constant across object sizes with default fusing. When fusing is off, the run time is 25% slower for 1 MB objects, and up to 12× slower when spilling 100 KB objects.

Prefetching Task Arguments. Ray prefetches task arguments in a pipelined manner so that arguments are ready on a worker by the time the task is scheduled. Figure 3.9 shows that pipelined fetching of task arguments reduces the run time by 60–80%, comparing with a baseline implementation that only starts fetching objects after the task is scheduled.

3.5.4.3 Fault Tolerance

To test fault recovery, we fail and restart a random worker node 30 seconds after the start of the run. This results in both executor failure and data loss, as the worker's local object store is also lost. In all cases, we rely on the distributed futures system to re-execute any lost tasks and to reconstruct any lost objects. Lineage reconstruction (§3.4.3.3) minimizes interruption time during worker failures. Figures 3.4a and 3.4b show run times with failures indicated with semi-shaded bars. For ES-simple and -merge, a known bug in Ray currently prevents fault recovery from completing. For ES-push and -push*, recovering from a worker failure adds 20–50 seconds to the job completion time. The system uses this time to detect node failures and re-execute tasks to reconstruct lost objects.

3.6 Related Work

Shuffle in Data Processing Systems. Many solutions to shuffle have been proposed [75, 38, 101, 48, 77, 76] since MapReduce [26] and Hadoop [9], with a focus on optimizing disk I/O and pipelining. Sailfish [75] is a notable example deployed at Yahoo which depends on a modified filesystem to batch disk I/O. Many recent shuffle systems have been built in industry for large-scale use cases [110, 16, 84, 13], but few have been open-sourced. Today's cloud providers often offer managed shuffle services [4, 85, 11]. However these are tightly integrated with proprietary cloud data services and are not accessible by other shuffle applications.

Hardware Environments. Hardware typically poses a range of constraints on shuffle design. For example, compute and memory may be either disaggregated [110, 16, 72] or colocated [75, 103, 84]. Disk constraints also affect system design, e.g., SSDs provide better random IOPS than HDDs but wear out more quickly. Many existing shuffle systems have been motivated by such hardware differences. In Exoshuffle, because the distributed futures API abstracts block management, a shuffle developer can plug in different storage backends and optimize shuffle at the application level.

Other Shuffle Applications. Machine learning research [57, 56] shows that SGD-based model training benefits from random shuffling of the training dataset. Both TensorFlow [62] and PyTorch [89] have built specialized systems designed specifically to pipeline data loading with ML training. These data loaders, in addition to Petastorm [37], support distributed data loading and random shuffling but shuffling is limited to a local buffer capped by worker memory (§3.5.3.3).

Dataframes [55, 79, 71] are another class of applications in data science that depend on shuffle for operations such as group-by. While systems like Dask [79] and Spark [105] provide distributed dataframes, developers continue to build new engines that optimize for specific application scenarios, such as multi-core [28], out-of-core performance [17], or supporting SQL [33]. These new dataframe libraries, along with new embedded query engines such as DuckDB [74] and Velox [70], can directly use Exoshuffle to support distributed query processing.

Distributed Programming Abstractions for Shuffle. CIEL [61] is the first to propose using distributed futures to express iterative distributed dataflow programs, including MapReduce. Its implementation does not include features critical to large-scale shuffle performance, including intra-node parallelism, in-memory object storage, and automatic garbage collection [63]. Dask [79] is another distributed futures-based system that has trouble scaling shuffle due to the lack of shared-memory objects (§3.5.4.1). While we build on Ray's design, such as a shared-memory object store [59] and lineage reconstruction [99], previous versions are not sufficient to support large-scale shuffle as they do not include spilling to disk or pipelining between execution and I/O. Thus, while others have implemented shuffle on distributed futures before, ours is the first that we know of to reach the scale, performance, and reliability of monolithic shuffle systems.

Serverless functions, as used in Locus [72], are one alternative to distributed futures. While Locus leverages an existing serverless cache and persistent storage, it still must manage block movement manually. In contrast, distributed futures abstract block management in full and manage execution, memory, and disk collectively on each node.

Hoplite [111] shows that it is possible to provide a high-performance and fault-tolerant collective communication layer on top of distributed futures, supporting operations such as scatter, gather, and reduce. Shuffle in MapReduce-like systems is a more challenging problem because it involves scheduling arbitrary compute tasks along with all-to-all communication. In this work we show that a distributed futures system can support shuffle at TB+ scale and provide competitive performance and reliability.

3.7 Discussion

Extensible Architectures. The decoupling of control and data planes in software-defined networking [54] has led to great innovations in the past two decades [32]. Operating systems

research also advocates for extensible architectures to build OS kernels, such as microkernels [104] and exokernels [31]. We hope our work can drive more innovations in shuffle designs and applications through an extensible architecture for distributed shuffle.

Distributed Futures. Distributed futures are rising in popularity due to their ease of use and flexibility [61, 59, 98]. However, the question of flexibility versus performance remains. Large-scale shuffle is one of the most challenging problems in big data processing, inspiring years of work. By showing that large-scale shuffle is possible on a generic and flexible distributed futures system, we hope to show that other complex applications can be built on this framework, too.

Limitations. The ability to specify arbitrary tasks and objects with distributed futures is the key to its flexibility, but it is also the primary obstacle to performance. The system assumes that each task is independent for generality and stores metadata separately for each task and object. In contrast, monolithic shuffle systems have semantic information and can share metadata for tasks and objects in the same stage. Currently metadata overhead is the main limitation to executing Exoshuffle at larger scales. We plan to address this in the future by "collapsing" shared metadata, i.e., keeping one metadata entry for multiple outputs of a task.

Architecturally, the primary limitation in Exoshuffle is the fact that an object must be loaded in its entirety into the local object store before it can be read (§3.4.3.1). Generators allow tasks to "stream" large outputs by breaking them into many smaller physical objects; future improvements include the described metadata optimizations and/or introducing APIs to stream objects larger than the object store, similar to Ciel [61]. Another limitation is in scheduling. Currently the distributed futures system may require hints from the shuffle library to determine which tasks should be executed concurrently and where to place tasks (§3.4.2). A more sophisticated scheduler may be able to determine these automatically.

Finally, Exoshuffle does not yet address the problem of providing a single shuffle solution that can meet the requirements of all applications. Doing so would require automatically picking the best shuffle algorithm and parameters based on application, environment, and run-time information. Instead, we focus on the problem of shuffle *evolvability*, a necessary step towards this overarching goal.

3.8 Conclusion

There is a longstanding tension between performance and flexibility in designing abstractions for distributed computing. Monolithic shuffle systems sacrifice flexibility in the name of

performance: they must essentially rebuild shuffle from scratch to handle varying application scenarios. In this work, we show that this need not be the case, by demonstrating an extensible architecture with a distributed futures system that makes it possible build efficient, flexible, and portable shuffle.

Chapter 4

Case Study: The CloudSort Benchmark

In this chapter, we present a case study of Exoshuffle-CloudSort, a sorting application running on Ray using the Exoshuffle architecture (Chapter 3). Sorting is one of the most challenging system performance benchmarks because it stress-tests all aspects of the system. To reach high performance on the sorting benchmark, the system must eliminate bottlenecks in both the hardware stack (CPU, memory, disk, network) and the software stack (OS, filesystem, runtime libraries). Through this case study, we demonstrate the scalability and performance of the Exoshuffle architecture for the most demanding distributed data processing jobs.

Exoshuffle-CloudSort runs on Amazon EC2, with input and output data stored on Amazon S3. Using 40× i4i.4xlarge workers, Exoshuffle-CloudSort completes the 100 TB CloudSort Benchmark (Indy category [82]) in 5378 seconds, with an average total cost of $97. In 2022, Exoshuffle-CloudSort set a new world record as the most cost-efficient sorting system on the public cloud. It is 33% more cost-efficient than the previous world record, set by Apache Spark in 2016 [95], and 15% cheaper when factoring in decreasing hardware costs.

This chapter is based on the published technical report [50] and the blog post [49], and includes contributions of Derek Luan, et al.

4.1 Design and Implementation

4.1.1 Overview

Exoshuffle-CloudSort is a distributed futures program running on top of Ray, a task-based distributed execution system. The program acts as the control plane to coordinate map and reduce tasks; the Ray system acts as the data plane, responsible for executing tasks, transferring blocks, and recovering from failures.

Exoshuffle-CloudSort implements a two-stage external sort algorithm. The first stage is map and shuffle. Each map task reads an input partition, sorts it, and partitions the result into W output partitions, each sent to a merger on a worker node. A merger receives W map output partitions, merges and sorts them, and further partition the result into R/W output partitions, all of which are spilled to local disk.

The second stage is reduce. Once the map and shuffle stage finishes, each reduce task reads W shuffled partitions, merges and sorts them, and writes the final output partition.

For the 100 TB CloudSort Benchmark, we set the following parameters:

- Total data size is 100 TB.
- Number of input partitions $M = 50\,000$. Each input partition is 2 GB.
- Number of workers $W = 40$.
- Number of output partitions $R = 25\,000$.

4.1.2 Preparation

The first step in Exoshuffle-CloudSort is to compute the partition boundary values. For a sort record with 10-byte key, we view the first 8 bytes as a 64-bit unsigned integer partition key. We partition the key space $[0, 2^{64} - 1)$ into $R = 25\,000$ equal ranges, such that all the records within a key range should be sent to one reducer.

Every $R_1 = R/W = 625$ reducer ranges are combined into a worker range, and records in each worker range will be sent to one worker node. This yields $W = 40$ equally-partitioned worker ranges.

4.1.3 Map and Shuffle Stage

In the map and shuffle stage, Exoshuffle-CloudSort schedules the $M = 50\,000$ map tasks onto all worker nodes. In our experiments we set the map parallelism, i.e. the number of map tasks running on a single worker node, to be 3/4 of the total number of vCPU cores. Extra tasks are queued on the driver node. Whenever a worker node finishes a map task, the driver assigns a new task from the queue to this node.

In a map task, we first download the input partition from S3. We then sort the input data in memory, then partition it into $W = 40$ slices. Each slice is eagerly sent to a merge controller on each worker. The map task returns when all slices are sent.

On the receiving end, the merge controller accumulates the map blocks in memory until a threshold is reached. We set the threshold to 40 blocks, or about 2 GB of data. Once

the threshold is reached, the controller launches a merge task to merge the already-sorted map blocks, and further partitions it into $R_1 = 625$ merged blocks, each corresponding to a reduce task on this node. These blocks are spilled to the local SSD for use by the reducers.

The merge parallelism is set to be the same as the map parallelism. When the number of merge tasks reaches the maximum parallelism, and the merge controller's in-memory buffer is filled up, it will hold off acknowledging the receipt of a map block until a merge task finishes and a new merge task can launch. This effectively creates back pressure to the map task scheduler to ensure the map, shuffle, and merge progresses are in sync.

In our experiments, the average map task duration is 24 seconds; 15 seconds are used for downloading input data. The average shuffle time (i.e. time to send and receive blocks) is 7 seconds. The merge task takes 17 seconds on average.

4.1.4 Reduce Stage

Once all map and merge tasks finish, Exoshuffle-CloudSort enters the reduce stage. Each reduce task loads $R_1 = 625$ from the local SSD, merges them, and uploads the sorted output partition to S3. In our experiments, each reduce task takes 22 seconds on average.

4.1.5 Handling Skewness

The previous steps work well assuming a uniformly distributed dataset. However, in a real-world situation, the dataset may often be skewed, which could cause a worker to be overloaded with data, run out of memory, and fail. We have implemented several techniques to handle skewness:

4.1.5.1 Sampling

We introduce a sampling step in the Preparation phase, where we sample a small portion of data points from each input part to create a subsample of the dataset. Then, we will generate partition boundaries such that it will split the subsample into evenly distributed sections. Next we will use these boundaries to partition our complete dataset. We have found that in most cases, using a sample of 20 random points from each partition creates an approximately uniform dataset from a skewed input.

4.1.5.2 Merging

Since sampling is an inherently random process, it is possible that the produced boundaries do not evenly split the data. If that happens, one or many workers will be overloaded with

data during the merge and reduce step and could run out of memory, crashing the worker and failing to get the final output.

We set a limit on the total data size that a merge task would merge on top of the existing limit on the number of blocks so that the merge task would not be overwhelmed by merging too much data at once.

4.1.5.3 Streaming

If the data is skewed, the input to a reduce task may be too large to process at once in worker memory. We implemented streaming in the reduce step so that no matter how large the input to a reduce task was, it would still complete successfully. This serves as a catch-all in case sampling is ineffective, and guarantees that Exoshuffle-Cloudsort will always generate the correct results.

To accomplish this, we track the size of every part, so that we know the size of the input to every reduce task (this was previously unknown as the input is passed in as a reference). Next, if the input to a reducer is greater than a certain size threshold, we launch a remote task to convert each part in the input into 100MB chunks which are stored back in the Ray Object Store. We set the threshold to be 4GB. Finally, we proceed with the reduce task and merge the parts using a lot less memory by streaming the chunks and only considering one relevant chunk of each part at a time. If the input is less than the size threshold, we process it without streaming since the complete input can fit within memory.

4.1.5.4 Evaluation

To evaluate the effectiveness of our solution to skewness, we ran our modified algorithm on 1 TB datasets in different scenarios using an Amazon EC2 cluster of 10× i4i.4xlarge instances.

As a baseline and sanity-check, we first ran the algorithm on a uniformly distributed 1 TB dataset. This took 292 seconds, which is close to the runtime of our original algorithm and shows that our additions only add a small overhead when the dataset is optimal.

To verify that our algorithm could truly handle unexpectedly large skewed inputs, we disabled the sampling step but kept the other changes and tested the algorithm on a skewed 1 TB dataset generated by gensort. We found that some reducers could receive inputs of around 256 GiB. Each node has 128 GiB total memory, which then must be split between multiple workers, so it is clear that the 256 GiB input would overwhelm the worker if handled naively. As expected, our algorithm did not crash or run out of memory and was able to handle the increased load to specific workers, completing successfully in 1820 seconds. The increased runtime is expected as the data was not evenly distributed between workers.

Finally, we tested the fully modified algorithm on a skewed 1 TB dataset generated by gensort. This took 295 seconds, which is only slightly more than our baseline. By integrating sampling, the data was close to evenly partitioned and our runtime massively improved.

4.1.6 The Execution System

A highlight of the Exoshuffle architecture is that the application program only implements the control plane logic, and the distributed futures system, Ray, handles execution. This is reflected in Exoshuffle-CloudSort. Here is an incomplete list of features provided by Ray that we take "for free":

- Task scheduling: The program specifies when and where to schedule tasks; the system handles the RPC, serialization, and other bookkeeping.
- Network transfer: The program instructs data to be transferred by passing distributed futures as task arguments; the system implements high-performance network transfer.
- Memory management and disk spilling: The program manipulates data references in a virtual, infinite address space; the system uses reference counting to manage distributed memory, spills objects to local disks when memory is low, and restores objects from local disks when they are needed.
- Pipelining of network and disk I/O: The network transfer, spilling and recovery of objects are transparent to the application and are performed asynchronously. For example, the system shuffles map output blocks while other map and merge tasks are running; it spills merge task output to disk while other merge tasks are executing, and it restores merged blocks while reduce tasks are executing.
- Fault tolerance: this is transparent to the application: the system automatically retries the operation when it encounters network failures and worker process failures.

For more details, we refer the reader to the Ray Architecture Whitepaper [91], the ownership design for distributed futures systems [99], and the Exoshuffle paper [51].

The execution pattern of Exoshuffle-CloudSort is known in the literature as *push-based* shuffle [84]. The main benefit is that it enables fine-grained pipelining of the transfer of blocks through network, memory, and disk. Figure 4.1 shows a pipelined view of the movement of an individual data partition, in which:

- Download and sort happens in the map tasks.
- Transfer happens in the background Ray workers.
- Merge happens in the merge tasks.
- Spill and restore happens in the background Ray workers.
- Reduce and upload happens in the reduce tasks.

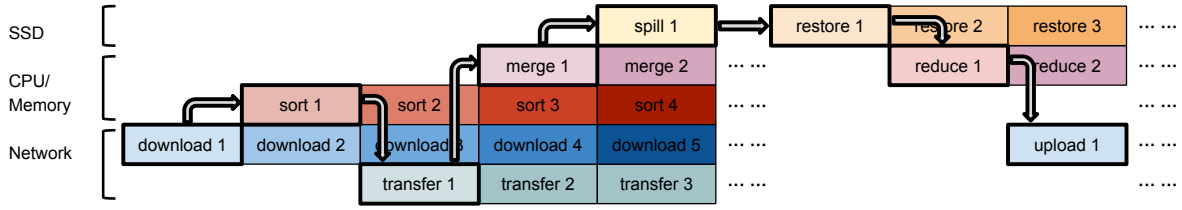


Figure 4.1: Pipelining of tasks in Exoshuffle-CloudSort.

4.1.7 Source Code

Exoshuffle-CloudSort is implemented in about 1000 lines of Python, and about 300 lines of C++. The C++ component implements two functionalities: sorting and partitioning records, and merging sorted record arrays. Exoshuffle-CloudSort runs on top of Ray, which is implemented in Python and C++. All of Exoshuffle-CloudSort's source code is available at `https://github.com/exoshuffle/cloudsort`.

4.2 Evaluation

4.2.1 Environment Setup

We run Exoshuffle-CloudSort on AWS on a compute cluster configured as follows:

- 1× r6i.2xlarge master node. This node runs on 8 cores of an Intel Xeon 8375C CPU at 2.9 GHz, and 64 GiB memory.
- 40× i4i.4xlarge worker nodes. Each node runs on 16 cores of an Intel Xeon 8375C CPU at 2.9 GHz, and 128 GiB memory. Each node has a directly-attached 3.75 TB AWS Nitro NVMe SSD.
- Each node is attached with a 40 GiB Amazon EBS General Purpose SSD (gp3) volume.

The software stack is configured as follows:

- Ubuntu 22.04.1 LTS, Linux kernel version 5.15.0-1022-aws.
- XFS 5.13.0 filesystem.
- Intel oneAPI DPC++/C++ Compiler 2022.2.0.20220730.
- Python 3.9.13.
- Ray 2.1.0.

We measure the raw system I/O performance on the worker nodes using standard benchmarking tools:

- Network bandwidth: 25 Gbps between nodes, benchmarked with iperf.
- SSD: 2.9 GB/s read, 2.2 GB/s write, benchmarked with fio.

For storage, we use 40 buckets on Amazon S3 and randomly distribute the input and output partitions across the buckets.

4.2.2 Benchmark Setup

Generating Input We use `gensort` version 1.5 as provided by the Sort Benchmark committee [65]. We run the command gensort −c −b{offset} {size} {path} to generate each partition. {size} is fixed at $P = 20\,000\,000$ such that each partition is exactly 2 GB. { offset } takes the values $\{i \cdot P : 0 \leq i < M\}$ where the number of input partitions $M = 50\,000$. {path} is a unique path in tmpfs. −c provides data checksum for validation. After generating an input file, we randomly choose a bucket and upload the partition to S3. We use Ray to schedule the 50 000 input generation tasks to all 40 worker nodes. The result is aggregated as an input manifest file, saved for use by Exoshuffle-CloudSort to locate the sort input.

Validating Output Exoshuffle-CloudSort produces an output manifest file containing the bucket and keys of each output partition on S3. In each validation task, we first download the output partition to tmpfs, then run the command valsort −o {sumpath} {path} to validate the ordering of records in each partition. We use Ray to schedule the 25 000 output validation tasks to all 40 worker nodes. We concatenate the contents of the summary files from each validation task, then run valsort −s to validate the total ordering, and generate the total output checksum. Finally, we compare the output checksum with the input checksum to verify data integrity.

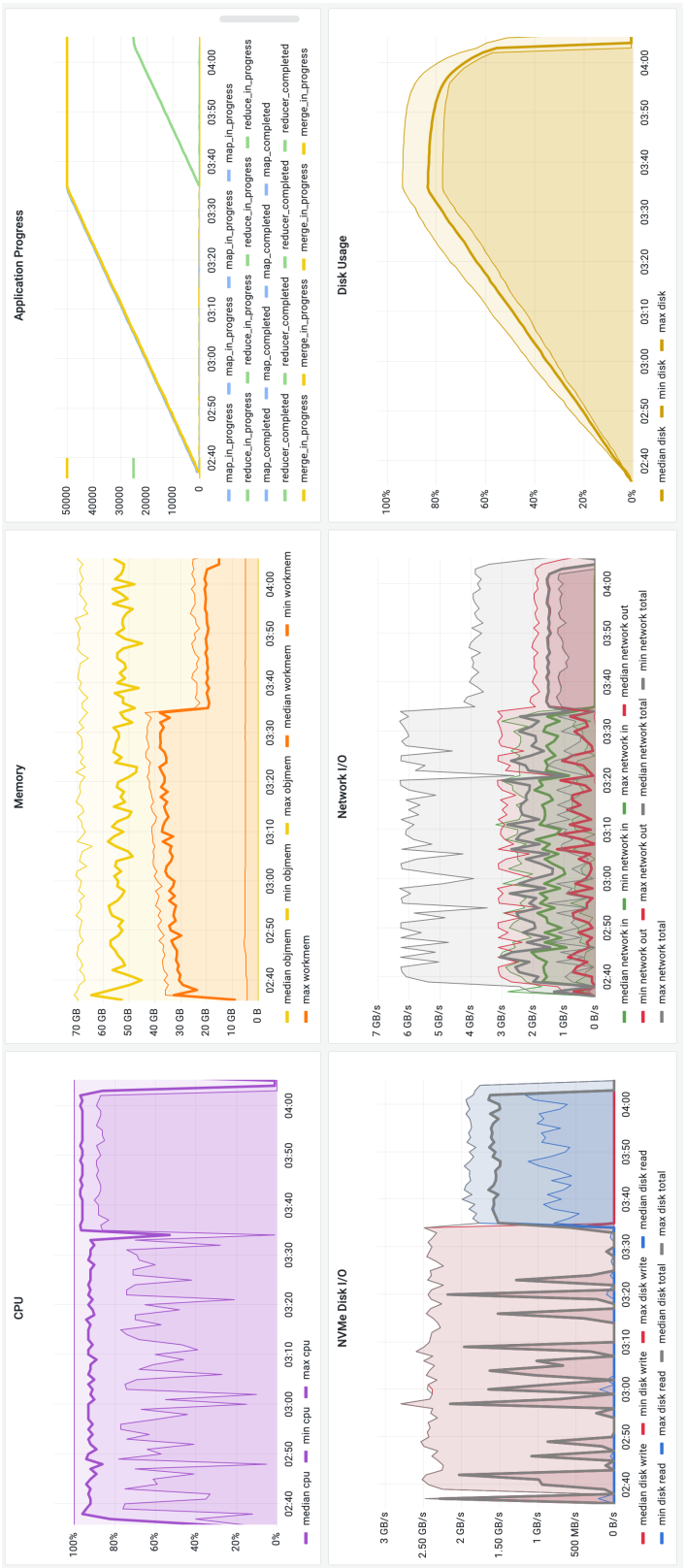


Figure 4.2: Cluster utilization during run #1 of the 100 TB CloudSort Benchmark. Each thick line represents the median system utilization of all worker nodes; the highest and lowest lines represent the maximum and minimum utilization among all worker nodes, respectively.

Run	Map & Shuffle Time	Reduce Time	Total Job Completion Time
#1	3509 s	1852 s	5361 s
#2	3496 s	1852 s	5348 s
#3	3520 s	1906 s	5426 s
Average	3508 s	1870 s	5378 s

Table 4.1: Job completion times of Exoshuffle-CloudSort on the 100 TB CloudSort Benchmark.

4.2.3 Experimental Results

4.2.3.1 Job Completion Time

On November 10, 2022, we ran the 100 TB CloudSort Benchmark in the AWS US West (Oregon, us-west-2) region with the setup described above. We first generated the input data on Amazon S3, then ran Exoshuffle-CloudSort 3 times, each followed by a validation step. All 3 runs succeeded with the same output checksum as the input, indicating all bytes are preserved in the sort. Table 4.1 reports the job completion times of each run. The average job completion time is 5378 seconds, or 1.4939 hours.

Figure 4.2 shows the system utilizations of all worker nodes in the compute cluster during run #1 of the 100 TB CloudSort Benchmark.

4.2.3.2 Total Cost of Ownership

The total job cost comprises of two parts: compute cost (Amazon EC2), and the storage cost (Amazon S3). The storage cost is further divided into data storage cost and data access cost.

Compute Cost The compute cost is calculated as the compute cluster's hourly cost times the job completion time. The total hourly cost is calculated as follows:

$$\begin{aligned}\text{Total Hourly Compute Cost} = {} & \text{Master Node Hourly Cost} \\ & + \text{Worker Node Hourly Cost} \times \text{Number of Workers} \\ & + \text{EBS Volume Hourly Cost} \times (\text{Number of Workers} + 1)\end{aligned} \tag{4.1}$$

We obtain the compute instance hourly costs from the Amazon EC2 on-demand pricing information [6]. For EBS, we use the Amazon EBS monthly price [5] divided by the average number of hours in a month ($\frac{365\times 24}{12} = 730$) as the hourly price. The hourly cost of a 40 GiB gp3 volume is $\$0.08/730 \times 40 = \$0.0044$. Now we plug the cost variables into Equation (4.1):

- Master node (r6i.2xlarge) hourly cost is $0.504.
- Worker node (i4i.4xlarge) hourly cost is $1.373.
- Number of workers is 40.
- EBS volume hourly cost is $0.0044.

Hence, the total hourly compute cost is $55.6044. We multiply this hourly cost by the job completion time of 1.4939 hours to obtain the total compute cost of $83.0674.

Data Storage Cost The storage cost comprises of data storage cost and data access cost. We first consider the data storage cost. Amazon S3 employs a pay-as-you-go pricing model, i.e. the user does not need to provision storage capacity ahead of time, and only pays for the storage cost of objects based on their sizes and storage duration. Amazon S3 charges $0.023 per GB-month for the first 50 TB, then $0.022 per GB-month for the next 450 TB [7]. Since the total data size is 100 TB, we take the average price between the first two tiers, i.e. $0.0225 per GB-month, or $3.0822 per hour per 100 TB.

- Input: The storage cost of the 100 TB input data is simply the cost to store 100 TB for the duration of the sort: $\$3.0822 \times 1.4939 = \$4.6045$.
- Output: The 100 TB output data is uploaded to and stored on Amazon S3 during the reduce stage of the sort. We use the duration of the reduce stage as the storage time of the 100 TB output data. This is an over-estimation because the output partitions are uploaded as the reduce stage progresses, and therefore most of the 100 TB is stored on S3 for less time than the entire reduce stage duration. Table 4.1 shows the average reduce stage time is 1870 seconds, or 0.5194 hours. Hence we get the output storage cost: $\$3.0822 \times 0.5194 = \$1.6009$.

Adding up the input and output data storage cost, we get the total data storage cost: $6.2054.

Data Access Cost We consider GET and PUT requests to Amazon S3. Exoshuffle-CloudSort downloads the 100 TB input data in 50 000 map tasks. Each map task downloads a 2 GB input partition in 16 MiB chunks, resulting in 120 GET requests per task, or 6 000 000 GET requests in total. Amazon S3 charges $0.0004 per 1000 GET requests [7]. Hence the total GET cost is $2.4000.

Exoshuffle-CloudSort uploads the output data in 25 000 reduce tasks. Each reduce task uploads approximately 4 GB data in 100 MB chunks, resulting in 40 PUT requests, or

1 000 000 PUT requests in total. Amazon S3 charges $0.005 per 1000 PUT requests [7]. Hence the total PUT cost is $5.0000.

The actual number of requests could be marginally higher due to request failures and retries, but the amount should be negligible. Hence, the total data access cost is $7.4000.

Total Cost of Ownership Adding up the compute cost and storage cost, we get the total cost of ownership for the 100 TB CloudSort Benchmark: $96.6728. Table 4.2 presents a summary of the cost analysis.

Service	Unit Price	Amount	Total Price
Compute VM Cluster	$55.6044 / hr	1.4939 hours	$83.0674
Data Storage (Input)	$3.0822 / hr	1.4939 hours	$4.6045
Data Storage (Output)	$3.0822 / hr	0.5194 hours	$1.6009
Data Access (Input)	$0.0004 / 1000 requests	6 000 000 requests	$2.4000
Data Access (Output)	$0.005 / 1000 requests	1 000 000 requests	$5.0000
Total	–	–	$96.6728

Table 4.2: Cost breakdown of Exoshuffle-CloudSort on the 100 TB CloudSort Benchmark.

4.3 Discussion

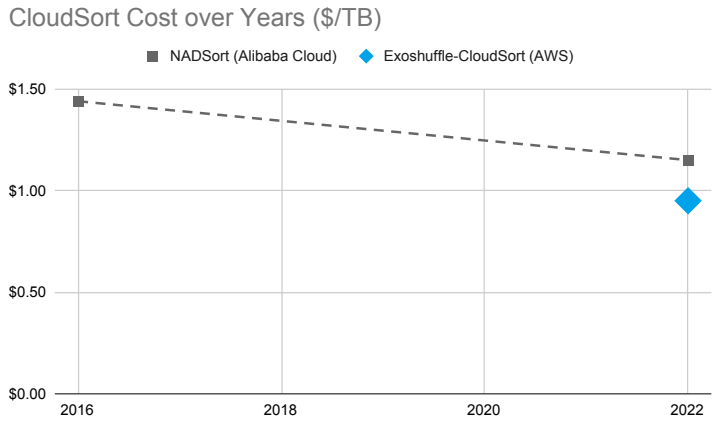


Figure 4.3: CloudSort cost over years.

Contributing Factors to the Reduced Cost The total cost of sorting 100 TB of data using Exoshuffle-CloudSort is 33% less than the previous world record, set using Apache Spark in 2016 [95]. One contributing factor is the lowering cost of compute and storage over years of technological advancements. However, this alone would not have made this new world record possible. As a baseline, we took the setup from the previous world record from 2016, and looked up its cost on today's Alibaba Cloud. It turned out that the same amount of resources that cost $144 in 2016 would cost $115 in 2022. Therefore, although the cloud is getting cheaper, the cost reduction does not fully explain the new record (Figure 4.3).

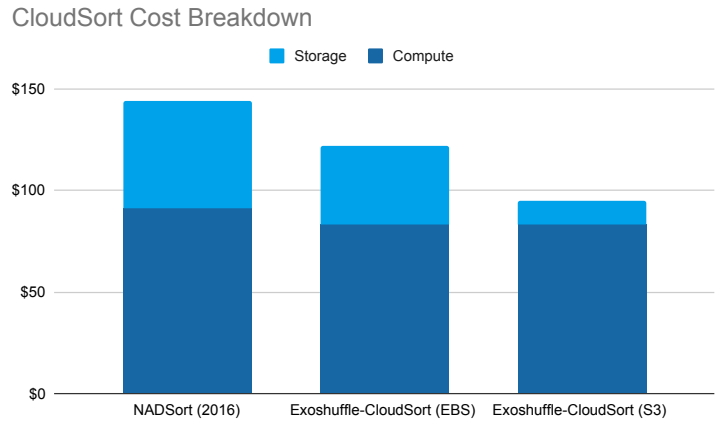


Figure 4.4: Cost breakdown of different CloudSort runs. Amazon S3 provides the best I/O performance with the lowest cost.

Disaggregated storage, the separation of compute and storage, turns out to be critical for breaking the CloudSort benchmark. Disaggregated storage allows for scaling compute and storage independently and elastically. This architecture is employed by recent data lake [10] and lakehouse [108] designs. In Exoshuffle, we store the input and output data on the storage service Amazon S3, instead of block storage devices mounted to individual VMs. Compared with traditional designs that colocate storage and compute (Figure 4.4), this architecture has several advantages:

1. S3 only charges for IOPS whereas EBS charges for both IOPS and throughput. For throughput-oriented applications such as sorting, S3 is therefore very cost-efficient. The S3 cost was $12 for CloudSort. If we used EBS instead, its cost would have been $38.74.

2. S3 storage does not require provisioning in advance, whereas EBS must be provisioned and mounted ahead of time. This makes S3 more elastic to varying data sizes and free of data locality issues.

3. Storing data on S3 means the data is preserved under worker failure. We are working to take advantage of this property, plus Ray's fault tolerance capabilities, to make CloudSort work on auto-scaling clusters of spot instances.

Conclusion The popular belief is that specialized shuffle systems are necessary for achieving high performance. The new world record set by Exoshuffle-CloudSort shows that this need not be the case: shuffle can run as an application program on a generic distributed execution system, such as Ray, with state-of-the-art performance and scalability. This opens up the possibility to run a rich set of data processing workloads on Ray.

Chapter 5

Conclusion and Future Directions

In this chapter, we first discuss new research directions unlocked by the work described in this thesis, both from the streaming batch execution model (§ 5.1) and from the Exoshuffle architecture (§ 5.2). Then, we draw conclusions from the research work presented in this dissertation.

5.1 Autoscaling Streaming Batch Processing

In Chapter 2, we showed that the streaming batch execution model is well-positioned to take advantage of elastically scaling resources. The main reason is that the adaptive scheduler constantly updates its view of the cluster resource utilization, and can therefore quickly adjust operator parallelisms to take advantage of the available resources.

Take batch inference as an example. As seen in Chapter 2, sometimes a single VM's I/O bandwidth or CPU processing power is not enough for the data loading and pre-processing stages, and as such, the inference stage on the GPU is bottlenecked by data. Figure 2.8 shows that adding an additional CPU-only VM could allieviate the bottleneck, such that the GPU can run at maximum throughput. The decision to scale out can be made automatically by the system. One possible design comprises:

- A mechanism to monitor the per-item throughput of each stage in the data processing pipeline (already exists in Ray Data).
- A cost model that estimates the throughput increase of a particular stage when adding more resources.
- A cluster manager that can automatically add or remove resources based on the cost model (already exists in Ray).
- A scheduler that can adaptively adjust the parallelism of each operator based on the (varying) available resources (already exists in Ray Data).

Such a design could enable truly autoscaling ML pipelines, allieviating the user and system administrators from the burden of estimating and pre-allocationg resources for a given job. The user would only need to specify the amount of available GPUs (as they are usually the scarcest resource), and the system would automatically scale out the other resources as needed to maximize the GPU utilization, and therefore the throughput of the pipeline.

5.2 Extending Exoshuffle to Petabyte Scale

In Chapter 3, we tested Exoshuffle on data analytics workloads up to 100 TB. In doing so, we observed performance trends that would prevent the current Ray-based implementation from scaling to petabyte-scale workloads. In this section, we discuss the challenges and potential solutions to scaling Exoshuffle to petabyte-scale workloads.

Exoshuffle uses Ray's shared-memory object store to track intermediate shuffle blocks. Each shuffle block is represented as a Ray object. Ray maintains an non-trivial amount of metadata for each object, including its owner, location(s), reference counts, and lineage information [99]. This metadata is stored in memory on the owner node of the object, and takes about 2 KB of memory per object.

Consider a 1 PB shuffle workload with 1 GB partitions. There will be 10^6 input partitions. The naive shuffle algorithm would create 10^{12} shuffle blocks, each represented as a Ray object. The metadata for these objects would consume 2 TB of memory, which is infeasible for a single node. Even though there are ways to shard the object metadata across multiple nodes, they will still take a significant amount of memory.

In the Ray object store, many associative containers, including the object location directory, and the object reference table, are implemented as flat hash tables (`absl::flat_hash_set`, `absl::flat_hash_map`). These hash tables contain pre-allocated contiguous memory regions, while at runtime, many of them are empty or contain only a single entry. It is possible to reduce their memory footprint with more memory-efficient data structures.

In the rest of this section, we describe two other opportunities for optimizing the Ray system to scale to petabyte-scale workloads.

5.2.1 Partial Object Access

In Ray, an object is an atomic unit of data that is either fully present or absent on a node. When a task requests an object, Ray fetches the entire object from the object store. This is inefficient when the task only needs a small portion of the object. For example, in a shuffle operation, each reduce task only needs to receive a small portion of the map partition. Passing a reference to the map partition to every reduce task will result in the entire object being transferred to each reduce task.

There is currently no way to express the correct semantics for partial object access in Ray. In Exoshuffle-CloudSort, this is implemented by dividing the map partition into many small objects, each representing a small portion of the map partition. This is inefficient because it increases the number of objects in the object store, despite that these objects share identical ownership and lineage information.

Proposed API Extend Ray's `ObjectRef` API with the subscript operator: if `x_ref` is an `ObjectRef`, then `x_ref[begin:end]` returns a `PartialObjectRef` that represents a slice of the original object. When resolving the partial object reference,

- If the object is already present on the node, then simply return the slice.
- If the object is in the memory of another node, then the system only fetches the sliced portion of the object, instead of the entire object.
- If the object is evicted from memory and spilled to disk, then the system restores only the sliced portion of the object from disk, and sends it to the requesting node.
- The slices count as references to the original object, and shares the lineage information with the original object.

This design will also benefit stream processing in Ray, which is implemented using streaming tasks. A streaming task yields a stream of small chunks, like a Python generator, instead of returning the result all at once. This reduces the peak memory usage of tasks that produce very large outputs. These chunks are similar to shuffle blocks in Exoshuffle, in that they also share the same owner and lineage. The implementation of streaming tasks could also be optimized using the partial object access API, which would result in much lower per-chunk metadata overhead.

5.2.2 Persisting Objects

In Exoshuffle, we extended Ray with the ability to spill objects to external storage (disk or cloud storage) when memory is full, and restore them into memory when they are needed again. This effectively establishes a tiered storage system that allows Ray applications to process datasets that are larger than the available memory. However, in the current implementation, only the object data can be spilled to external storage, while the object metadata must remain in memory.

This design has undesirable implications for failure recovery: if the object metadata is lost due to a node failure, then the object data becomes irrecoverable, despite it being intact in the external storage.

Proposed Feature Serialize and persist the object metadata to external storage, in addition to the object content data. In the event of an owner node failure, the system can retrieve the object metadata from the external storage, and then reconstruct the object in memory from the serialized content. This will reduce the amount of lineage reconstruction and recomputation required.

Besides persisting object metadata, it could also be beneficial to improve the object spilling mechanism with features commonly found in production-grade data processing systems in the industry. For example, many systems (such as HDFS [9]) allow users to set the replication factor of the spilled objects. This will allow disk-based clusters to be more resilient to node failures, as the system can restore the lost objects from the other replicas.

5.3 Conclusion

In this dissertation, I have argued that *extensibility*—the ability to adapt to evolving hardware capabilities and application requirements—is the key principle in the design of distributed heterogeneous processing system through two key contributions: the streaming batch execution model and the Exoshuffle architecture. The streaming batch model enables efficient execution of heterogeneous pipelines by dynamically adapting to varying workload characteristics and resource availability. The Exoshuffle architecture demonstrates that complex distributed operations like shuffle can be implemented as application libraries without sacrificing performance or scalability. Both of these work made real-world impact as part of the open-source Ray Data framework for scalable ML data pre-processing, which has been adopted by thousands of companies worldwide. As the scale and complexity of data processing workloads continue to grow, and the landscape of heterogeneous hardware continues to evolve, the principle of extensibility will be essential for building systems that can efficiently scale to support the next generation of data processing applications.

Bibliography

[1] Sami Abu-El-Haija et al. *YouTube-8M: A Large-Scale Video Classification Benchmark*. 2016. arXiv: 1609.08675 [cs.CV]. URL: https://arxiv.org/abs/1609.08675.

[2] Tyler Akidau et al. "Millwheel: Fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[3] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803. ISSN: 2150-8097. DOI: 10.14778/2824032.2824076. URL: https://doi.org/10.14778/2824032.2824076.

[4] Alibaba. *EMR Remote Shuffle Service: A Powerful Elastic Tool of Serverless Spark - Alibaba Cloud Community*. https://www.alibabacloud.com/blog/emr-remote-shuffle-service-a-powerful-elastic-tool-of-serverless-spark_597728. (Accessed on 02/01/2022). May 2021.

[5] Amazon. *Amazon EBS High-Performance Block Storage Pricing*. Amazon Web Services. 2022. URL: https://aws.amazon.com/ebs/pricing/ (visited on 11/10/2022).

[6] Amazon. *Amazon EC2 On-Demand Instance Pricing*. Amazon Web Services. 2022. URL: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 11/10/2022).

[7] Amazon. *Amazon S3 Simple Storage Service Pricing*. Amazon Web Services. 2022. URL: https://aws.amazon.com/s3/pricing/ (visited on 11/10/2022).

[8] Apache Software Foundation. *Apache Beam*. https://beam.apache.org. Apache Software Foundation, May 2024.

[9] Apache Software Foundation. *Hadoop*. https://hadoop.apache.org. Version 3.3.1. Apache Software Foundation, June 2021.

[10] Michael Armbrust et al. "Delta lake: high-performance ACID table storage over cloud object stores". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3411–3424. ISSN: 2150-8097. DOI: 10.14778/3415478.3415560. URL: https://doi.org/10.14778/3415478.3415560.

[11] Anubhav Awasthi, Rajendra Gujja, and Mohit Saxena. *Introducing Amazon S3 shuffle in AWS Glue*. `https://aws.amazon.com/blogs/big-data/introducing-amazon-s3-shuffle-in-aws-glue/`. (Accessed on 10/16/2022). Nov. 2021.

[12] Johannes Bader et al. "AI in Software Engineering at Facebook". In: *IEEE Software* 38.4 (2021), pp. 52–61. DOI: 10.1109/MS.2021.3061664.

[13] Mayank Bansal and Bo Yang. *Zeus: Uber's Highly Scalable and Distributed Shuffle as a Service - Databricks*. `https://databricks.com/session_na20/zeus-ubers-highly-scalable-and-distributed-shuffle-as-a-service`. (Accessed on 02/01/2022). July 2020.

[14] Jon C. R. Bennett and Hui Zhang. "WF2Q: worst-case fair weighted fair queueing". In: *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies Conference on The Conference on Computer Communications - Volume 1*. INFOCOM'96. San Francisco, California: IEEE Computer Society, 1996, pp. 120–128. ISBN: 0818672927.

[15] Juraj Bienik et al. "Performance of H.264, H.265, VP8 and VP9 Compression Standards for High Resolutions". In: *2016 19th International Conference on Network-Based Information Systems (NBiS)*. 2016, pp. 246–252. DOI: 10.1109/NBiS.2016.70.

[16] Dmitry Borovsky and Brian Cho. *Cosco: An Efficient Facebook-Scale Shuffle Service - Databricks*. `https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service`. (Accessed on 01/19/2022). May 2019.

[17] Maarten A. Breddels and Jovan Veljanoski. "Vaex: big data exploration in the era of Gaia". In: *Astronomy & Astrophysics* 618 (Oct. 2018), A13. DOI: 10.1051/0004-6361/201732493. URL: https://doi.org/10.1051/0004-6361/201732493.

[18] Tim Brooks, Aleksander Holynski, and Alexei A. Efros. "InstructPix2Pix: Learning To Follow Image Editing Instructions". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2023, pp. 18392–18402.

[19] Lingfeng Cai et al. "Failure-resilient DAG task scheduling in edge computing". In: *Computer Networks* 198 (Aug. 2021), p. 108361. DOI: 10.1016/j.comnet.2021.108361.

[20] Jose Cambronero et al. "When deep learning met code search". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 964–974. ISBN: 9781450355728. DOI: 10.1145/3338906.3340458. URL: https://doi.org/10.1145/3338906.3340458.

[21] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[22] Kelvin CK Chan et al. "Investigating tradeoffs in real-world video super-resolution". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 5962–5971.

[23] Eugenia Cheng. *How to Bake Pi: An Edible Exploration of the Mathematics of Mathematics*. Reprint. Basic Books, May 2016, p. 304. ISBN: 978-0465097678.

[24] Tyson Condie et al. "MapReduce Online". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI '10. San Jose, California: USENIX Association, 2010, p. 21.

[25] Jason Jinquan Dai et al. "BigDL: A Distributed Deep Learning Framework for Big Data". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 50–60. ISBN: 9781450369732. DOI: 10.1145/3357223.3362707. URL: https://doi.org/10.1145/3357223.3362707.

[26] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[27] A. Demers, S. Keshav, and S. Shenker. "Analysis and simulation of a fair queueing algorithm". In: *Symposium Proceedings on Communications Architectures & Protocols*. SIGCOMM '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 1–12. ISBN: 0897913329. DOI: 10.1145/75246.75248. URL: https://doi.org/10.1145/75246.75248.

[28] Polars Developers. *Polars – User Guide*. https://pola-rs.github.io/polars-book/user-guide/index.html. (Accessed on 10/16/2022). 2022.

[29] Spark developers. *Spark Release 3.2.0*. https://spark.apache.org/releases/spark-release-3-2-0.html. (Accessed on 01/26/2022). Oct. 2021.

[30] E. N. (Mootaz) Elnozahy et al. "A survey of rollback-recovery protocols in message-passing systems". In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408. ISSN: 0360-0300. DOI: 10.1145/568522.568525. URL: https://doi.org/10.1145/568522.568525.

[31] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076. URL: https://doi.org/10.1145/224056.224076.

[32] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 87–98. ISSN: 0146-4833. DOI: 10.1145/2602204.2602219. URL: https://doi.org/10.1145/2602204.2602219.

[33] Apache Software Foundation. *Apache Arrow DataFusion Documentation*. https://arrow.apache.org/datafusion/. (Accessed on 10/16/2022). 2022.

[34] Yossi Gandelsman et al. "Test-Time Training with Masked Autoencoders". In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=SHMi1b7sjXk.

[35] Ali Ghodsi et al. "Multi-resource fair queueing for packet processing". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 2012, pp. 1–12.

[36] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[37] Robbie Gruener, Owen Cheng, and Yevgeni Litvin. *Introducing Petastorm: Uber ATG's Data Access Library for Deep Learning*. https://www.uber.com/blog/petastorm/. Uber, Sept. 2018.

[38] Yanfei Guo, Jia Rao, and Xiaobo Zhou. "iShuffle: Improving Hadoop Performance with Shuffle-on-Write". In: *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 107–117. ISBN: 978-1-931971-02-7. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/guo.

[39] Ajay Gupta. *Revealing Apache Spark Shuffling Magic*. https://medium.com/swlh/revealing-apache-spark-shuffling-magic-b2c304306142. (Accessed on 02/01/2022). May 2020.

[40] Sasun Hambardzumyan et al. *Deep Lake: a Lakehouse for Deep Learning*. 2022. arXiv: 2209.10785 [cs.DC]. URL: https://arxiv.org/abs/2209.10785.

[41] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[42] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation". In: *SIGMOD Rec.* 26.2 (June 1997), pp. 171–182. ISSN: 0163-5808. DOI: 10.1145/253262.253291. URL: https://doi.org/10.1145/253262.253291.

[43] Tarun Kalluri et al. "Flavr: Flow-agnostic video representations for fast frame interpolation". In: *arXiv preprint arXiv:2012.08512* (2020).

[44] James Knighton et al. *MosaicML StreamingDataset: Fast, Accurate Streaming of Training Data from Cloud Storage*. https://www.databricks.com/blog/mosaicml-streamingdataset. Mosaic AI Research, Feb. 2023.

[45] Peter Kraft et al. "Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference". English (US). In: *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys)*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 147–159.

[46] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. Athens, Greece. 2011, pp. 1–7.

[47] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[48] Jingui Li et al. "Improving the Shuffle of Hadoop MapReduce". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. Bristol, UK: IEEE, 2013, pp. 266–273. DOI: 10.1109/CloudCom.2013.42.

[49] Frank Sifei Luan. *Ray breaks the $1/TB barrier as the world's most cost-efficient sorting system*. Anyscale. 2023. URL: https://www.anyscale.com/blog/ray-breaks-the-usd1-tb-barrier-as-the-worlds-most-cost-efficient-sorting/.

[50] Frank Sifei Luan et al. *Exoshuffle-CloudSort*. 2023. DOI: 10.48550/ARXIV.2301.03734. URL: http://sortbenchmark.org/ExoshuffleCloudSort2022.pdf.

[51] Frank Sifei Luan et al. "Exoshuffle: An Extensible Shuffle Architecture". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 564–577. ISBN: 9798400702365. DOI: 10.1145/3603269.3604848. URL: https://doi.org/10.1145/3603269.3604848.

[52] Sifei Luan et al. "Aroma: code recommendation via structural code search". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360578. URL: https://doi.org/10.1145/3360578.

[53] Peter Mattson et al. "MLPerf Training Benchmark". In: *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys)*. Vol. 2. Austin, TX, USA, 2020, pp. 336–349.

[54] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[55] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. Austin, Texas: Python in Science Conference, 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[56] Qi Meng et al. "Convergence Analysis of Distributed Stochastic Gradient Descent with Shuffling". In: *Neurocomput.* 337.C (Apr. 2019), pp. 46–57. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.01.037. URL: https://doi.org/10.1016/j.neucom.2019.01.037.

[57] Konstantin Mishchenko, Ahmed Khaled, and Peter Richtarik. "Random Reshuffling: Simple Analysis with Vast Improvements". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Virtual: Curran Associates, Inc., 2020, pp. 17309–17320. URL: https://proceedings.neurips.cc/paper/2020/file/c8cc6e90ccbff44c9cee23611711cdc4-Paper.pdf.

[58] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. *Ludwig: a type-based declarative deep learning toolbox*. 2019. DOI: 10.48550/ARXIV.1909.07930. URL: https://arxiv.org/abs/1909.07930.

[59] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/moritz.

[60] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 439–455.

[61] Derek G. Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 113–126.

[62] Derek G. Murray et al. "Tf.Data: A Machine Learning Data Processing Framework". In: *Proc. VLDB Endow.* 14.12 (July 2021), pp. 2945–2958. ISSN: 2150-8097. DOI: 10.14778/3476311.3476374. URL: https://doi.org/10.14778/3476311.3476374.

[63] Derek Gordon Murray. "A distributed execution engine supporting data-dependent control flow". PhD thesis. University of Cambridge, 2012.

[64] Arvind Neelakantan et al. "Text and Code Embeddings by Contrastive Pre-Training". In: *CoRR* abs/2201.10005 (2022). arXiv: 2201.10005. URL: https://arxiv.org/abs/2201.10005.

[65] Chris Nyberg. *Sort Benchmark Data Generator and Output Validator*. Ordinal Technology Corp. 2022. URL: http://www.ordinal.com/gensort.html (visited on 11/10/2022).

[66] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

[67] A.K. Parekh and R.G. Gallager. "A generalized processor sharing approach to flow control in integrated services networks: the single-node case". In: *IEEE/ACM Transactions on Networking* 1.3 (1993), pp. 344–357. DOI: 10.1109/90.234856.

[68] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Red Hook, NY, USA: Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[69] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[70] Pedro Pedreira et al. "Velox: Meta's Unified Execution Engine". In: *Proc. VLDB Endow.* 15.12 (2022), pp. 3372–3384. URL: https://www.vldb.org/pvldb/vol15/p3372-pedreira.pdf.

[71] Devin Petersohn et al. *Towards Scalable Dataframe Systems*. 2020. DOI: 10.48550/ARXIV.2001.00888. URL: https://arxiv.org/abs/2001.00888.

[72] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[73] PyTorch. *torch.utils.data – PyTorch 2.3 documentation*. 2024. URL: https://pytorch.org/docs/stable/data.html.

[74] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. ISBN: 9781450356435. DOI: 10.1145/3299869.3320212. URL: https://doi.org/10.1145/3299869.3320212.

[75] Sriram Rao et al. "Sailfish: A Framework for Large Scale Data Processing". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391233. URL: https://doi.org/10.1145/2391229.2391233.

[76] Alexander Rasmussen et al. "Themis: An I/O-Efficient MapReduce". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391242. URL: https://doi.org/10.1145/2391229.2391242.

[77] Alexander Rasmussen et al. "TritonSort: A Balanced Large-Scale Sorting System". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 29–42.

[78] Vijay Janapa Reddi et al. "MLPerf inference benchmark". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 446–459. ISBN: 9781728146614. DOI: 10.1109/ISCA45697.2020.00045. URL: https://doi.org/10.1109/ISCA45697.2020.00045.

[79] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra. Austin, Texas: Python in Science Conference, 2015, pp. 130–136.

[80] Robin Rombach et al. "High-resolution image synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 10684–10695.

[81] Saksham Sachdev et al. "Retrieval on source code: a neural code search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 31–41. ISBN: 9781450358347. DOI: 10.1145/3211346.3211353. URL: https://doi.org/10.1145/3211346.3211353.

[82] Mehul A. Shah, Amiato, and Chris Nyberg. *CloudSort: A TCO Sort Benchmark*. http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf. (Accessed on 11/10/2022). June 2014.

[83] Min Shen. *RPC implementation to support pushing and merging shuffle blocks*. https://issues.apache.org/jira/browse/SPARK-32915. (Accessed on 10/16/2022). Sept. 2020.

[84] Min Shen, Ye Zhou, and Chandni Singh. "Magnet: Push-Based Shuffle Service for Large-Scale Data Processing". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3382–3395. ISSN: 2150-8097. DOI: 10.14778/3415478.3415558. URL: https://doi.org/10.14778/3415478.3415558.

[85] Sergei Sokolenko. *How Distributed Shuffle improves scalability and performance in Cloud Dataflow pipelines*. https://cloud.google.com/blog/products/data-analytics/how-distributed-shuffle-improves-scalability-and-performance-cloud-dataflow-pipelines. Sept. 2018.

[86] Lirong Song. *Dear Child: Awakening and Liberation of Life*. Chinese. Beijing: Line Arts Bureau, Sept. 2023. ISBN: 978-7-5120-5534-6.

[87] Yu Sun et al. "Test-time training with self-supervision for generalization under distribution shifts". In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, 2020.

[88] Gemini Team et al. "Gemini: a family of highly capable multimodal models". In: *arXiv preprint arXiv:2312.11805* (2023).

[89] PyTorch Team. *torch.utils.data – PyTorch documentation*. https://pytorch.org/docs/stable/data.html. (Accessed on 10/16/2022). 2022.

[90] Ray Team. *Ray Datasets: Distributed Data Preprocessing*. https://docs.ray.io/en/latest/data/dataset.html. (Accessed on 10/16/2022). 2022.

[91] Ray Team. *Ray v2 Architecture*. Anyscale. 2022. URL: https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI/preview (visited on 11/10/2022).

[92] Zhan Tong et al. "VideoMAE: Masked Autoencoders are Data-Efficient Learners for Self-Supervised Video Pre-Training". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 10078–10093. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/416f9cb3276121c42eebb86352a4354a-Paper-Conference.pdf.

[93] Joseph Torres et al. *Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3*. Mar. 2018. URL: https://www.databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html.

[94] Shivaram Venkataraman et al. "Drizzle: Fast and Adaptable Stream Processing at Scale". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 374–389. ISBN: 9781450350853. DOI: 10.1145/3132747.3132750. URL: https://doi.org/10.1145/3132747.3132750.

[95] Qian Wang et al. *NADSort*. http://sortbenchmark.org/NADSort2016.pdf. (Accessed on 01/26/2022). 2016.

[96] Stephanie Wang. *Analyzing memory management and performance in Dask-on-Ray*. https://medium.com/distributed-computing-with-ray/analyzing-memory-management-and-performance-in-dask-on-ray-930a2236b70d. (Accessed on 01/26/2022). June 2021.

[97] Stephanie Wang. "Towards a Distributed OS for Data-Intensive Cloud Applications". PhD thesis. EECS Department, University of California, Berkeley, Jan. 2024. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-3.html.

[98] Stephanie Wang, Benjamin Hindman, and Ion Stoica. "In Reference to RPC: It's Time to Add Distributed Memory". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 191–198. ISBN: 9781450384384. DOI: 10.1145/3458336.3465302. URL: https://doi.org/10.1145/3458336.3465302.

[99] Stephanie Wang et al. "Ownership: A Distributed Futures System for Fine-Grained Tasks". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. Virtual: USENIX Association, Apr. 2021, pp. 671–686. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/cheng.

[100] Xintao Wang et al. "EDVR: Video Restoration With Enhanced Deformable Convolutional Networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2019.

[101] Yandong Wang et al. "JVM-Bypass for Efficient Hadoop Shuffling". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Cambridge, MA, USA: IEEE, 2013, pp. 569–578. DOI: 10.1109/IPDPS.2013.13.

[102] Zhanghao Wu et al. "Can't Be Late: Optimizing Spot Instance Savings under Deadlines". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 185–203.

[103] Reynold Xin. *Apache Spark the Fastest Open Source Engine for Sorting a Petabyte*. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html. (Accessed on 01/19/2022). Oct. 2014.

[104] M. Young et al. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System". In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP '87. Austin, Texas, USA: Association for Computing Machinery, 1987, pp. 63–76. ISBN: 089791242X. DOI: 10.1145/41457.37507. URL: https://doi.org/10.1145/41457.37507.

[105] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[106] Matei Zaharia et al. "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud'12. Boston, MA: USENIX Association, 2012, p. 10.

[107] Matei Zaharia et al. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 2.

[108] Matei A. Zaharia et al. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *Conference on Innovative Data Systems Research*. 2021. URL: https://api.semanticscholar.org/CorpusID:229576171.

[109] Kai Zeng et al. "G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 913–918. ISBN: 9781450327589. DOI: 10.1145/2723372.2735381. URL: https://doi.org/10.1145/2723372.2735381.

[110] Haoyu Zhang et al. "Riffle: Optimized Shuffle Service for Large-Scale Data Analytics". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190534. URL: https://doi.org/10.1145/3190508.3190534.

[111] Siyuan Zhuang et al. "Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 641–656. ISBN: 9781450383837. DOI: 10.1145/3452296.3472897. URL: https://doi.org/10.1145/3452296.3472897.