The Streaming Batch Model for Efficient and Fault-Tolerant Heterogeneous Execution

Frank Sifei Luan[1] Ziming Mao[1] Ron Yifeng Wang[1] Charlotte Lin[1] Amog Kamsetty[1, *]
Hao Chen[2] Cheng Su[2] Balaji Veeramani[2] Scott Lee[2] SangBin Cho[2] Clark Zinzow[3, *]
Eric Liang[1, *] Ion Stoica[1, 2] Stephanie Wang[2, 4]

[1]*UC Berkeley* [2]*Anyscale* [3]*Together AI* [4]*University of Washington*

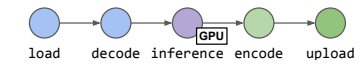
Abstract

While ML model training and inference are both GPU-intensive, CPU-based data processing is often the bottleneck. Distributed data processing systems based on the batch or stream processing models assume homogeneous resource requirements. They excel at CPU-based computation but either under-utilize heterogeneous resources or impose high overheads on failure and reconfiguration. We introduce the *streaming batch* model, a hybrid of the two models that enables efficient and fault-tolerant heterogeneous execution. The key idea is to execute one *partition* at a time to allow lineage-based recovery with dynamic resource allocation. This enables memory-efficient pipelining across heterogeneous resources, similar to stream processing, but also offers the elasticity and fault tolerance properties of batch processing. We present Ray Data, an implementation of the streaming batch model that improves throughput on heterogeneous batch inference pipelines by 3–8× compared to traditional batch and stream processing systems. When training Stable Diffusion, Ray Data matches the throughput of single-node ML data loaders while additionally leveraging distributed heterogeneous clusters to further improve training throughput by 31%.

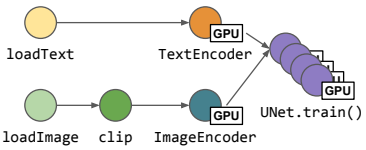
1 Introduction

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 [30] and batch inference [22]. Meanwhile, as ML models evolve, the data processing functionality required has also become more diverse, spanning many modalities, transformations, and resource requirements [19, 32, 41].

There are numerous frameworks designed to scale out CPU-based data processing, typically based on either the batch [5, 11, 47] or stream [2, 9, 23, 28] processing models. These systems allow users to express a dataflow of logical *operators* (Figure 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 [30].



(a) Batch inference.



(b) Training Stable Diffusion.

Figure 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.

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 [16, 32, 38, 40], 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 [46]. Ideally, a CPU failure should have little impact on GPU execution. This is challenging to achieve simultaneously with exactly-once record processing and minimal run-time overheads. Even with stateless transforms, the system must track which records have been processed for each operator to avoid

*Work done while at Anyscale.

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 [11, 47]. 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, 28] or high recovery overhead from global checkpointing and rollback [9, 28].

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 [27] 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.
- Ray Data, a fault-tolerant, memory-efficient, and autotuning implementation of the streaming batch model.

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.

2.1 Applications

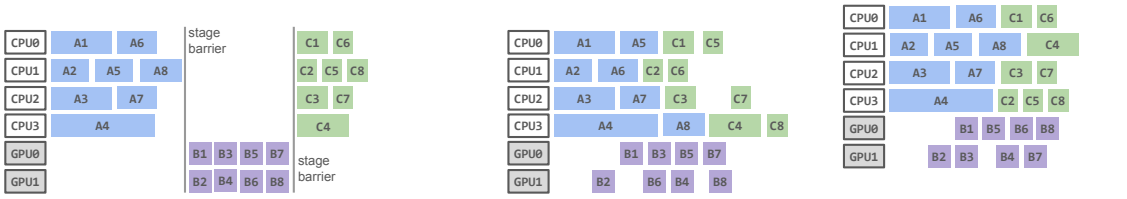
We target ML training and inference pipelines that require data pre- and post-processing using CPUs, GPUs, or both. Our goal is maximizing throughput, while providing sub-second latency when used as data loaders for ML training [30]. Similar to current ML dataloaders [30, 35], we primarily target map-style 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 [25].

Figure 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 1b shows a typical distributed training pipeline for the Stable Diffusion model [38]. 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 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 1a) into a physical DAG of stages of data-parallel tasks (Figure 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. Elas-



(a) Batch processing model. (b) Stream processing model. (c) Streaming batch model (Ray Data).

Figure 2: Execution models for Figure 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.

tic scaling is supported by simply adding or removing executors. Examples include MapReduce [11], Apache Hadoop [5], Apache Spark [47], Spark Streaming [48], and Apache Flink in `BATCH` execution mode [9].

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 [47]. This is effective when operators require the same resources. For example, suppose the `load` and `filter` operators in Figure 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 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 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 1a were run on a dataset of 100 videos, each 10MB on disk but 20GB decoded [7]. 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 [47], again before execution. Thus, it is left to the user to predict each operator's memory usage, then manually configure the number of partitions.

2.3 Stream Processing Model

Stream processing architectures optimize for online scenarios and asynchronous execution. Examples include Naiad [28], Apache Flink [9], Spark Continuous Processing [42], MillWheel [2], and Apache Kafka [23]. 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 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 2b, `A8` is pre-assigned to CPU3. In contrast, Figure 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 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 [43], and especially so for heterogeneous pipelines. First, for heterogeneous operators, fusion is often impractical. For example, Figure 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* [9, 28]. 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 [13]. Thus, reconfiguration typically introduces long pauses, as it requires taking then restarting from a global checkpoint with the new configuration [9].

Other systems use *logging* [2, 23]. 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 [30] and PyTorch DataLoader [35]. 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 [30].

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 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.

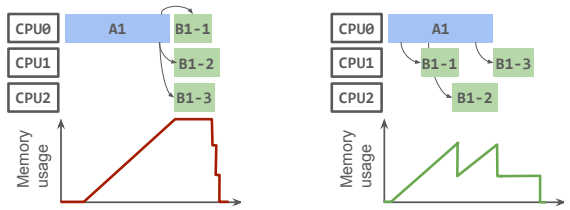
3 Overview: The Streaming Batch Model

The streaming batch model (Figure 2c) uses task-based execution with a centralized scheduler. Tasks can run on any executor, similar to batch processing (Figure 2a). However, tasks across different stages are pipelined and dynamically repartitioned (Figure 3b), similar to stream processing systems (Figure 2b). Table 1 shows a full feature comparison.

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.

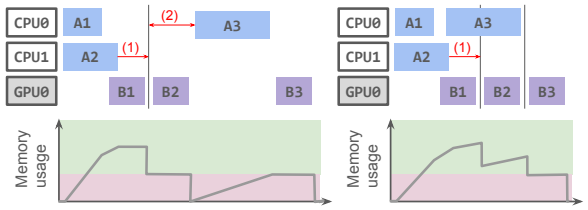
Challenge 1: Fault-tolerant, memory-efficient partitioning. Batch processing tasks are often memory-inefficient when operators require different parallelisms. For example, Figure 3a shows a two-stage pipeline where parallelism=1 and 3 for stages `A` and `B`, respectively. Repartitioning `A1` requires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



(a) Repartitioning in batch processing. (b) Streaming repartition.

Figure 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 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.

To address this, we propose a *streaming repartition* (§ 4.2.1): a task may dynamically generate multiple output partitions, based on its real-time memory consumption. For example, in Figure 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 [44] (§ 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 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

	Batch [5, 11, 47]	Stream [2, 9, 23, 28]	PyTorch DL [35]	tf.data [30]	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 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 (§ 2.3).

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

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 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 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 4b). Applying such optimizations requires dynamic profiling and reconfiguration.

To address this challenge, we introduce an adaptive scheduler in Section 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.

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). 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

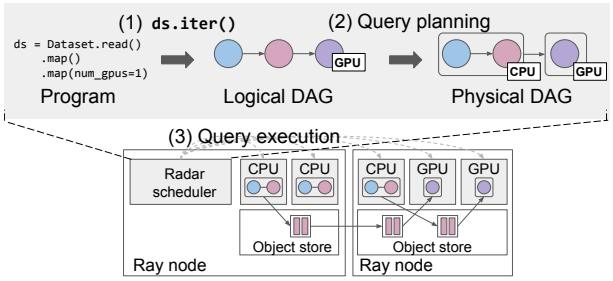


Figure 5: Ray Data architecture overview. Ray Data executes as a Ray library. The Ray Data scheduler executes as a Ray driver, dispatching tasks to Ray workers (dashed arrows).

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.

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 5. The `Dataset` is represented as a DAG of logical operators, as in Figure 1. The system's query planner then compiles this logical DAG into a DAG of physical operators ((2) in Figure 5). The query planner applies operator fusion and decides the number of partitions to use for the first operator (§ 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 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 [44]. 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 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 [44]. If a materialized partition is lost due to node failure, Ray automatically recreates it by resubmitting the tasks in its lineage. In Section 4.2.1, we describe the extensions we made to Ray's recovery method to support streaming repartition.

4 System Design

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) 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.

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 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.

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 (Figure 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.

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 [44], 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 [9, 30, 34, 47], 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, 9].

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.

4.3.1 Algorithm

The input to the scheduler is as follows:

- The physical DAG of operators (§ 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 $totalMemoryCapacity$ of the system.

Algorithm 1 describes the scheduling loop in step (3) of Figure 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 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 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 4b. To achieve this, lines 4–8 add a higher-priority optimistic policy for scheduling source tasks, described further in the next section.

Algorithm 1 Adaptive Scheduler

```
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 ← arg min_{op∈Q} bufferedOutputsSize(op)
19:         Launch task of selected
20:     end if
21: end while
```

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.

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

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

Algorithm 2 Memory Budget Update (runs every second)

$P \leftarrow 0$ ▷ Total processing time per partition
$\alpha_0 \leftarrow 1$ ▷ $\alpha_i :=$ Input:Output size ratio for op_i
for $i \leftarrow 1$ to $numOps$ **do**
  $E_i \leftarrow \text{availableExecutionSlots}(op_i)$
  $T_i \leftarrow \text{estimatedTaskDuration}(op_i)$
  if op_i is not *source* **then**
    $I_i \leftarrow \text{estimatedInputSize}(op_i)$
    $O_i \leftarrow \text{estimatedOutputSize}(op_i)$
    $\alpha_i \leftarrow \alpha_{i-1} \cdot O_i / I_i$
  end if
  $P_i \leftarrow (T_i/E_i) \cdot \alpha_{i-1}$
  $P \leftarrow P + P_i$
end for
$budget \leftarrow budget + \text{outputPartitionSize}(source)/P$

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 ~3s, the budget will be replenished to allow for one more source task to run.

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 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.

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 [44], and automatic disk spilling [25]. 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 ~90k Python LoC, including ~45k LoC for the query planner, scheduler and executor logic.

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 [26, 36], and supplement with additional image-to-image, video-to-video, and video classification benchmarks. We aim to answer:

- § 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?
- § 5.2: How does distributed and heterogeneous execution improve on throughput per dollar compared to ML-specific single-node data loaders?
- § 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 2a): Apache Spark 3.5.1.
- Stream processing (Figure 2b): Apache Flink 1.19.0.
- Single-node stream processing: tf.data [30]. 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) [35]. Similar to above, but uses multiprocessing and requires manual fusing of all operators.
- Streaming batch: Ray Data (implemented over Ray 2.40.0). We also modify Ray Data to emulate batch processing (Ray Data-staged) and stream processing (Ray Data-static), for apples-to-apples comparison of the execution models in § 2. Ray Data-staged materializes each stage before starting the next, while Ray Data-static sets a static parallelism per operator and disables the adaptive scheduler described in § 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.

5.1 Inference: Comparison to distributed batch and stream processing

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 [8] image dataset from S3, (2) `decode`: decode JPEG images to tensors, (3) `preprocess`: normalize tensors, (4) `Img2ImgModel`: use Stable Diffusion [38] 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.

```
radar.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 6a 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 ~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 Ray Data-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. Ray Data-dynamic is the default system, with the adaptive scheduler (§ 4.3). This automatically converges to the same throughput as Ray Data-static.

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 (Ray Data-static), with no configuration needed.

5.1.2 Video-to-Video Generation

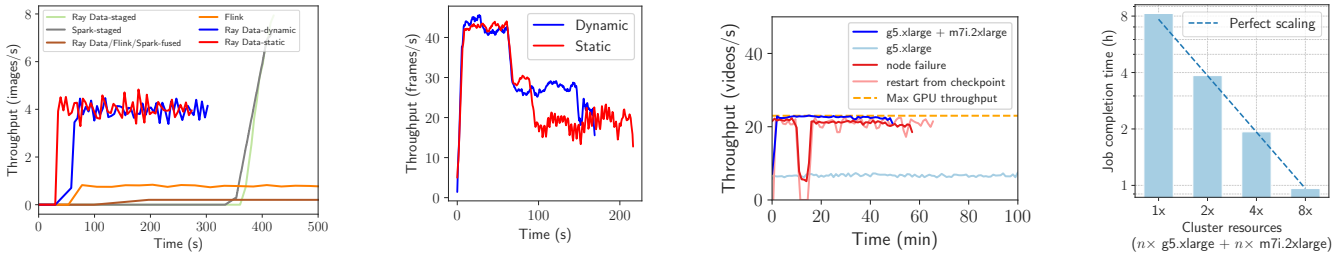
Video-to-video generation uses a generative model for use cases including super-resolution, interpolation, or deblurring [10, 20, 45]. Compared to § 5.1.1, video generation adds significant memory pressure: each video decodes to many frames. Also, high workload variability is common due to varying video 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 [10] 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 6b compares the throughput over time of Ray Data-static and Ray Data-dynamic, the two best systems in § 5.1.1. Ray Data-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 Ray Data-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.

5.1.3 Fault tolerance in heterogeneous clusters

We first demonstrate Ray Data's ability to scale with heterogeneous clusters. We run the VideoMAE [41] 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 § 5.1.2 except with less postprocessing. Data is stored on Amazon S3. Figure 6c 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



(a) Image generation. (b) Video generation. (c) Fault tolerance. (d) Scalability.

Figure 6: (a) Image generation throughput comparison. Radar-dynamic (blue) can match the performance of hand-tuned Radar-static (red) with static parallelism, and outperform other systems. *-staged execute synchronously; no results are available until the last stage begins at ~330s. (b) Video generation throughput comparison. Radar-dynamic (blue) achieves 28% better throughput compared to Radar-static (red) with static parallelism. (c) Fault tolerance comparison. Radar 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. (d) Scalability analysis. Radar is able to achieve strong scaling with respect to the number of nodes.

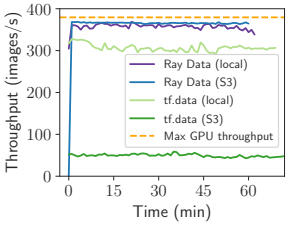


Figure 7: Training ResNet-50.

	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%)**

Figure 8: Run time and cost for one epoch of Stable Diffusion pre-training.

scaling the cluster up to 16 nodes, or 8 GPUs and 96 vCPUs in total. Figure 6d 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 6c 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.

5.2 Training: Comparison to ML data loaders

5.2.1 ResNet Training

We run the ResNet-50 ImageNet training benchmark from MLPerf [26]. 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 [30].

Figure 7 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 (§ 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.

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 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.

Figure 8 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 precomputed 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.

5.3 Microbenchmarks

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 9 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.

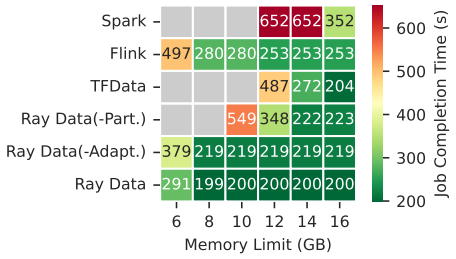


Figure 9: Synthetic benchmark run times for systems under different memory limits. Grey means the system is unable to finish due to OOM. Ray Data(-Part.) means Ray Data without streaming repartition. Ray Data(-Adapt.) means Ray Data without adaptive memory-aware scheduling.

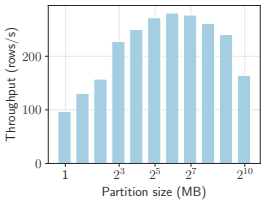


Figure 10: Effect of partition sizes on throughput in Ray Data.

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), 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 (§ 2.3). At lower memory limits, Flink must run fewer executors because it uses multithreading and slot sharing to multiplex a CPU slot among physical operators, 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 3b) and adaptive scheduler (§ 4.3). We further conduct ablation studies on Radar's performance. Ray Data(-Part.) disables Ray Data's streaming repartition, resulting in too-large initial partitions similar to Spark. Ray Data(-Adapt.) disables Ray Data's adaptive scheduler, resulting in a conservative policy similar to Figure 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.

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 10 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.

6 Related Work

Unifying batch and stream processing systems. Recent efforts to unify the batch and stream processing models include Apache Beam [4] 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 [28], Flink's batch execution mode [9], and Spark Streaming [47]. 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 [43] 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 [17] 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 [27, 44], CIEL [29] and Dask [37]. 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 Section 4.3 is most similar to the generalized processor sharing [33] problem. Our solution is inspired by the weighted fair queueing algorithm [6, 12]. 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 [15] addresses the problem of (2) but not (1) or (3). tf.data [30] 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 [34] and TensorFlow (tf.data [30]) 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 [21] and DeepLake [18]. 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.

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 § 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 [25].

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 (§ 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 [31], GPU-based embedding computation, and joins with image or video data (Figure 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 [14, 39], retrieval-augmented generation [24], and multimodal models [32, 40]. To keep up with this demand, we must build more flexible, heterogeneity-aware, and scalable data processing systems.

References

[1] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark, 2016.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, aug 2015.

[4] Apache Software Foundation. Apache beam. https://beam.apache.org, May 2024.

[5] Apache Software Foundation. Apache hadoop. https://hadoop.apache.org, May 2024.

[6] 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, page 120–128, USA, 1996. IEEE Computer Society.

[7] Juraj Bienik, Miroslav Uhrina, Michal Kuba, and Martin Vaculik. 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)*, pages 246–252, 2016.

[8] 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)*, pages 18392–18402, June 2023.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[10] Kelvin CK Chan, Shangchen Zhou, Xiangyu Xu, and Chen Change Loy. Investigating tradeoffs in real-world video super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5962–5971, 2022.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols*, SIGCOMM '89, page 1–12, New York, NY, USA, 1989. Association for Computing Machinery.

[13] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[14] Yossi Gandelsman, Yu Sun, Xinlei Chen, and Alexei A Efros. Test-time training with masked autoencoders. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[15] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 1–12, 2012.

[16] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis,

Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi,

Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024.

[17] Robbie Gruener, Owen Cheng, and Yevgeni Litvin. Introducing petastorm: Uber atg's data access library for deep learning. https://www.uber.com/blog/petastorm/, September 2018.

[18] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, and Davit Buniatyan. Deep lake: a lakehouse for deep learning, 2022.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[20] Tarun Kalluri, Deepak Pathak, Manmohan Chandraker, and Du Tran. Flavr: Flow-agnostic video representations for fast frame interpolation. *arXiv preprint arXiv:2012.08512*, 2020.

[21] James Knighton, Karan Jariwala, Davis Blalock, and Erica Ji Yuen. Mosaicml streamingdataset: Fast, accurate streaming of training data from cloud storage. https://www.databricks.com/blog/mosaicml-streamingdataset, February 2023.

[22] Peter Kraft, Daniel Kang, Deepak Narayanan, Shoumik Palkar, Peter Bailis, and Matei Zaharia. Willump: A statistically-aware end-to-end optimizer for machine learning inference. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys)*, volume 2, pages 147–159, 2020.

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

[24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[25] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi

Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark. In *Proceedings of the 3rd Conference on Machine Learning and Systems (MLSys)*, volume 2, pages 336–349, Austin, TX, USA, 2020.

[27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[28] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[29] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. {CIEL}: A universal execution engine for distributed {Data-Flow} computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[30] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: a machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[31] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. Text and code embeddings by contrastive pre-training. *CoRR*, abs/2201.10005, 2022.

[32] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B.

Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[33] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[35] PyTorch. torch.utils.data – pytorch 2.3 documentation, 2024.

[36] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[37] Matthew Rocklin et al. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, pages 126–132, 2015.

[38] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[39] Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. 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.

[40] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[41] Zhan Tong, Yibing Song, Jue Wang, and Limin Wang. Videomae: Masked autoencoders are data-efficient learners for self-supervised video pre-training. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 10078–10093. Curran Associates, Inc., 2022.

[42] Joseph Torres, Michael Armbrust, Tathagata Das, and Shixiong Zhu. Introducing low-latency continuous processing mode in structured streaming in apache spark 2.3, March 2018.

[43] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 374–389, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for {Fine-Grained} tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686, 2021.

[45] Xintao Wang, Kelvin CK Chan, Ke Yu, Chao Dong, and Chen Change Loy. Edvr: Video restoration with enhanced deformable convolutional networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 0–0, 2019.

[46] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: Optimizing spot instance savings under deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 185–203, 2024.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster

computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[48] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 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, page 10, USA, 2012. USENIX Association.

A Additional Microbenchmarks

A.1 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 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 11 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.

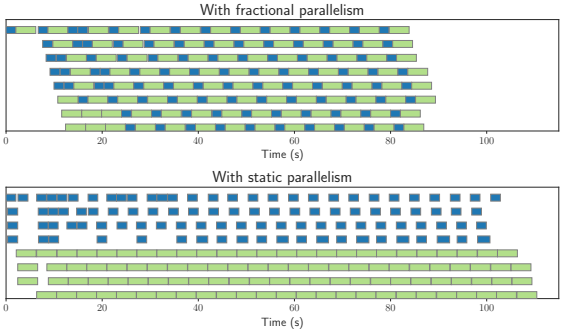


Figure 11: Dynamically allocated executor slots can achieve fractional parallelism with better resource utilization (fewer bubbles).

B 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.

B.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 Section 5.3.1, the solver finds the optimal schedule with a total run time of 153 seconds.