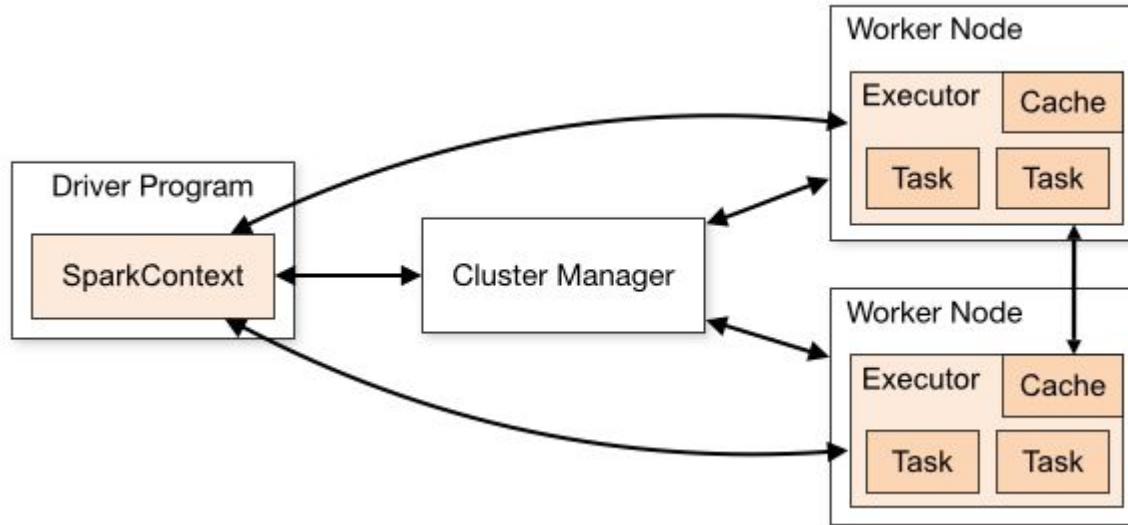


Last Time

- Introduced Parallel Processing
- Looked at Parallel Dataflow as a common set of operations that can be readily parallelized
- Studied parallel join and parallel aggregation
- Introduced Dask, a parallel implementation of Pandas

Spark Architecture



Spark Operations

- RDD: Resilient Distributed Dataset, a collection of elements that can be operated in parallel
 - Data flows in the DAG in the form of RDD
- Transformation: produce new RDDs given input RDDs
 - map, join, union, filter ...
 - Lazy evaluation: building the dataflow DAG
- Action: executes the DAG and returns results to driver program
 - Count, persist, take ...
- Demo

Spark Fault Tolerance Model

- Lineage-based recomputation
- When a worker node failed during execution, only the RDD partitions that are affected are recomputed.



Lecture 17: Scalable Data Processing with



Slides courtesy of Prof. Stephanie Wang

Website: stephanie-wang.github.io
Email: smwang@cs.washington.edu

About Professor Wang 🙌

- Incoming assistant professor at UW CSE
- Software engineer at Anyscale
- Lead author and committer of the Ray project, created at UC Berkeley



anyscale



Outline

1. What is Ray?
 - a. History of open-source project and system architecture
2. Ray Data deep dive
3. What's next for Ray?

What Problem is Ray Trying to Address?

Trends:

1. AI compute demands exploding → Need **scale**
2. AI application diversity exploding → Need **flexibility**
 - a. Diversity in data(sets)
 - b. Diversity in compute needs

Can We Use Spark or Dask for This?

AI workloads have flavor(s) of parallel execution supported by Spark/Dask

- Feature extraction
- Last-mile data-loading and preprocessing (i.e. data streaming)
- Model inference
- Model training

Q: could you write a Spark program to do each of these^ individually?

Note: Spark MLLib and Dask-ML support training decision trees & random forests

Case Study: Reinforcement Learning (RL)

RL 30,000 foot overview (one training epoch):

1. Init. NN model (called “**policy**”)
2. Policy is provided an initial state (e.g. chessboard, sensor data)
3. Policy is asked to take an **action**
4. Action is simulated in an environment
5. Environment returns new state (i.e. “**observation**”) and a **reward**
6. Steps 2-5 repeat for T time steps to produce a “trajectory”
7. N trajectories are produced and used to train / update the policy model

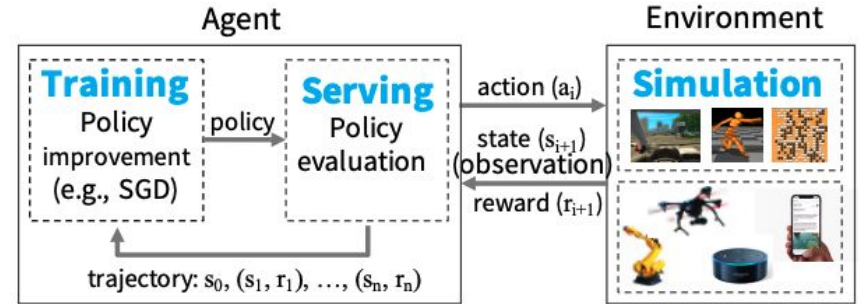


Figure 1: Example of an RL system.

Do not need to know this for quiz

Case Study: Reinforcement Learning (RL)

This AI workload requires us to support:

1. Model Serving
2. Parallel (distributed, possibly asynchronous) simulation
3. Model Training

Can we write a Spark / Dask program which does this? Is it efficient? Why or why not?

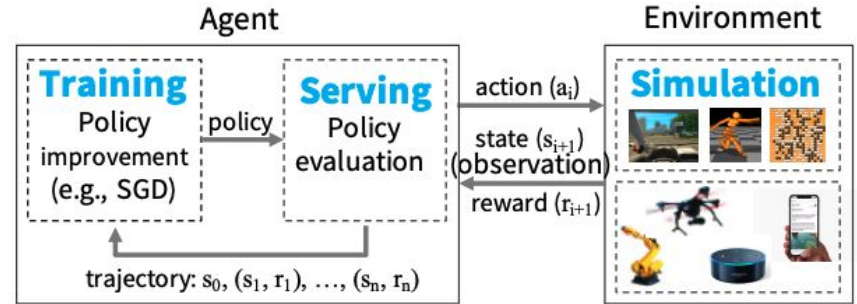


Figure 1: Example of an RL system.

What More Might We Want / Need for AI Workloads?

In a nutshell: fine-grained, low-level control over compute and data placement

- Spark and Dask are a bit too coarse-grained and synchronous
 - E.g. “run the same transformation over different partitions of a dataset (likely using homogenous hardware)”
 - Great fit for many data processing workloads
 - But possibly too optimized to be general purpose enough for AI workloads
- Ray offers lower-level programming interface which is ideal for these workloads
 - E.g. “co-locate my policy model and simulation environment on GPU instances, but run the simulation code on a separate set of CPU-optimized instances”
 - Support for stateful execution (Actors) and stateless execution (Tasks) in Ray Core
 - Easy to build optimized ML libraries and pipelines on top of this

Notably, Ray is reported to have been OpenAI’s framework of choice for training GPT

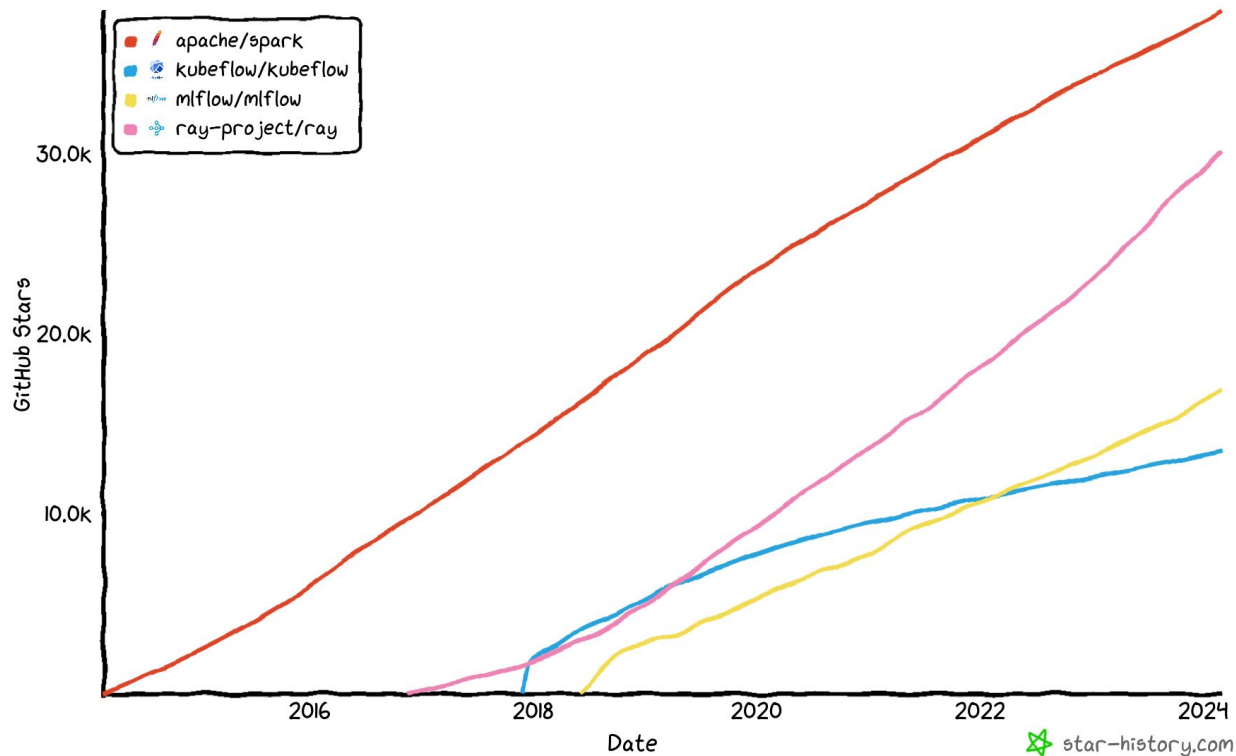
History of Ray

- 2016: Started development of v0.1 at UC Berkeley, in the RISELab
 - First version in Rust: <https://github.com/amplab/orchestra>
 - C++ prototype with gRPC: <https://github.com/ray-project/ray-legacy>
 - C prototype
 - May 2017: [v0.1 released](#)

History of Ray

- 2016: Started development of v0.1 at UC Berkeley, in the RISELab
- 2017: Tune (hyperparameter search) and RLlib (reinforcement learning) libraries
- 2018: Rewrite Ray core in C++; first Ray paper at OSDI'18
- 2019: Anyscale founded; began second rewrite of Ray core
- 2020: Ray v1.0 released; first Ray Summit; Serve (ML serving) library
- 2021: Ray v1.0 paper at NSDI'21; Ray Data
- 2022: Ray v2.0; OpenAI releases ChatGPT
- 2023: Ray beats Spark on CloudSort world record

GitHub star history



Ray: A Unified System for ML

ML libraries

Hyperparameter
Search

Distributed
Training

Simulation

Inference

Data
Processing

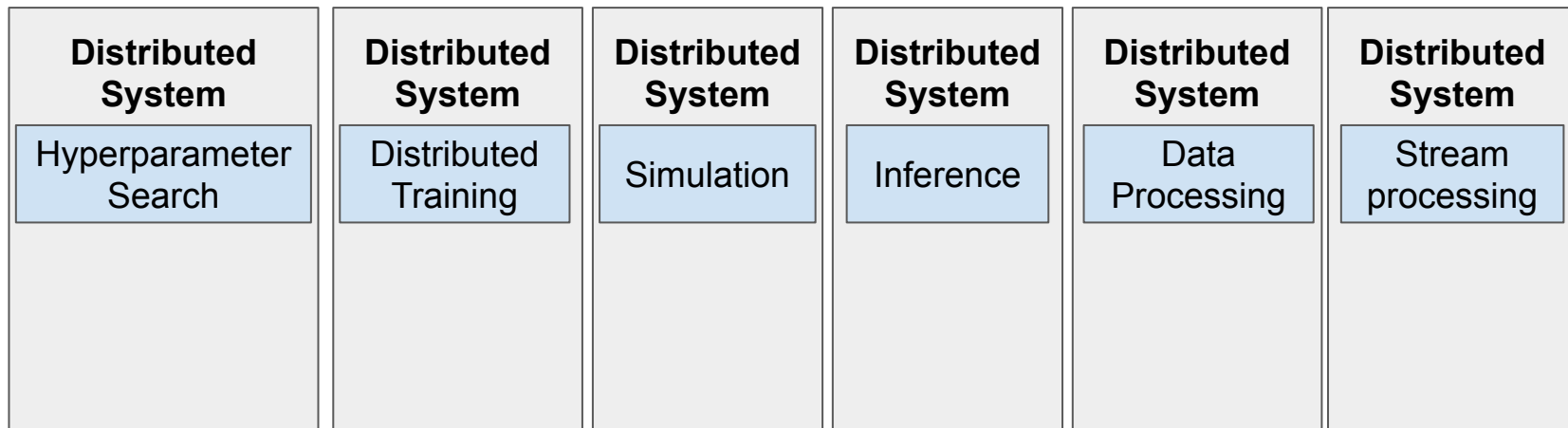
Stream
processing

On a single node, Python libraries are the key to app development:

- + **Performance:** Libraries often optimized with native code.
- + **Developer productivity:** Easily compose libraries with function calls.

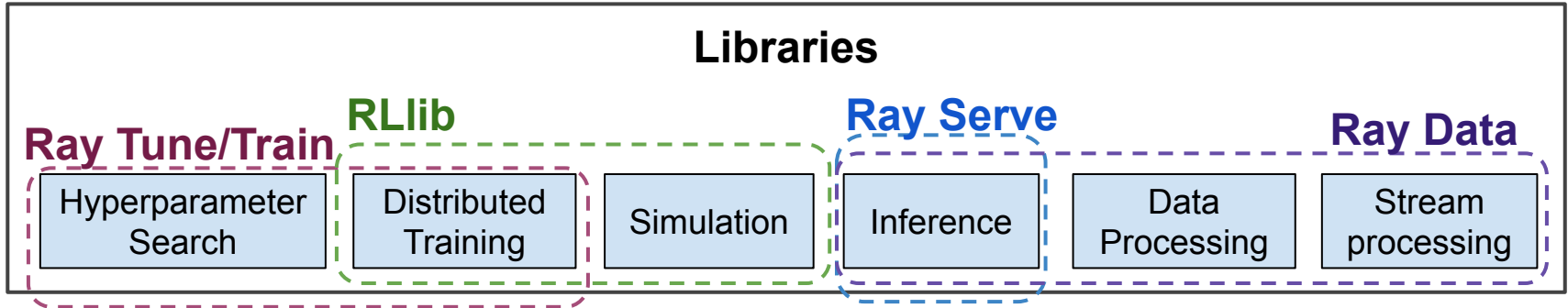
Problem: In the distributed setting, need to address domain-specific problems in scheduling, fault tolerance, etc.

Ray: A Unified System for ML



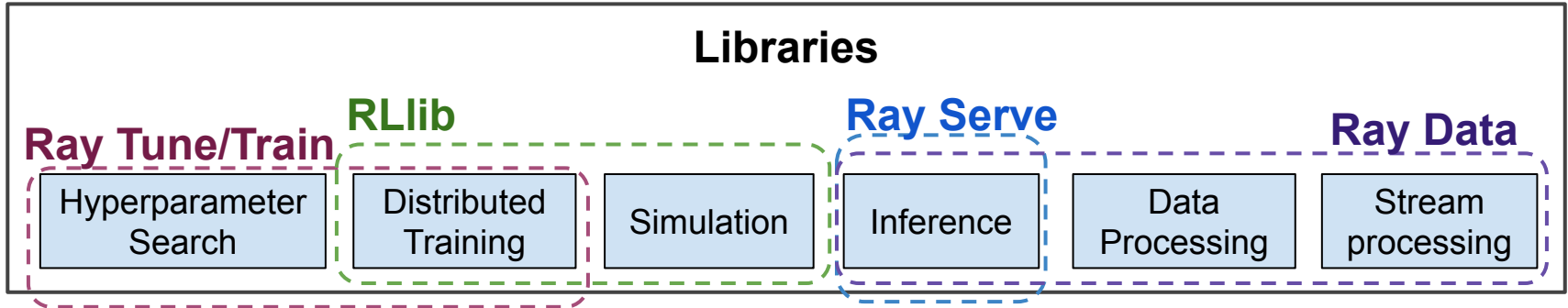
- **Developer productivity:** Orchestration? Data movement?
- **Performance:** End-to-end performance? Future-proof systems?

Ray: A Unified System for ML



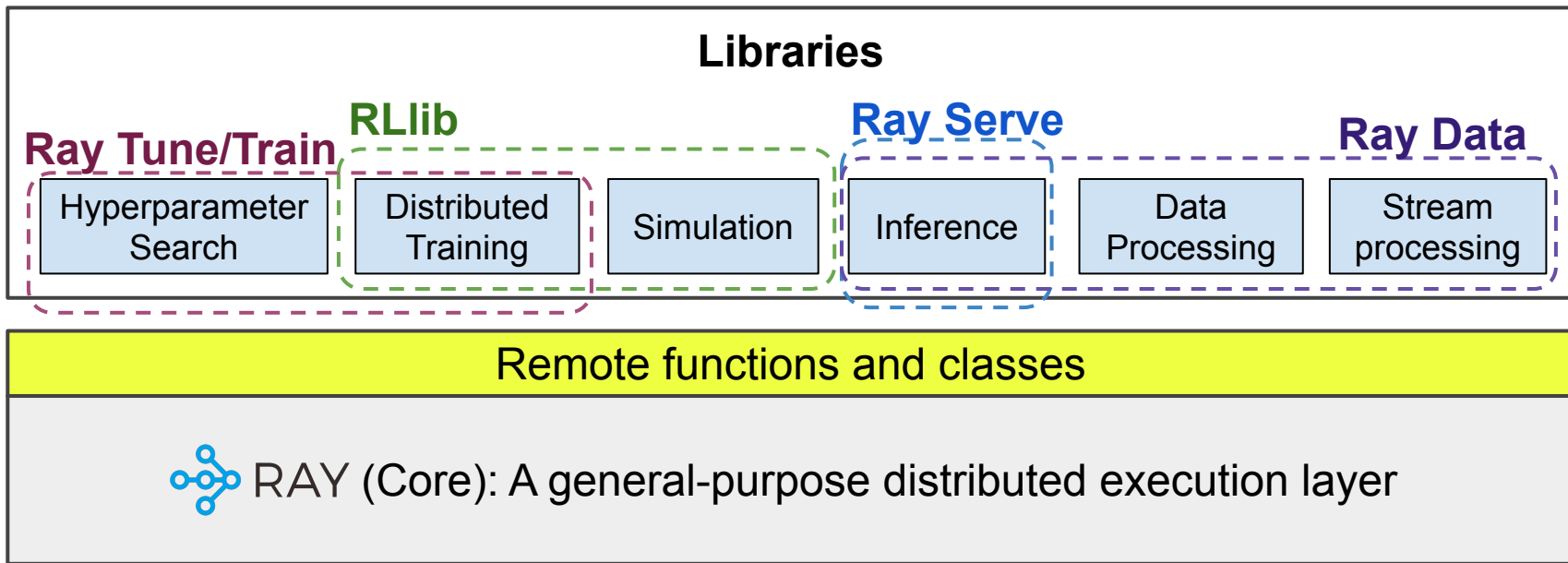
RAY (Core): A general-purpose distributed execution layer

Ray: A Unified System for ML



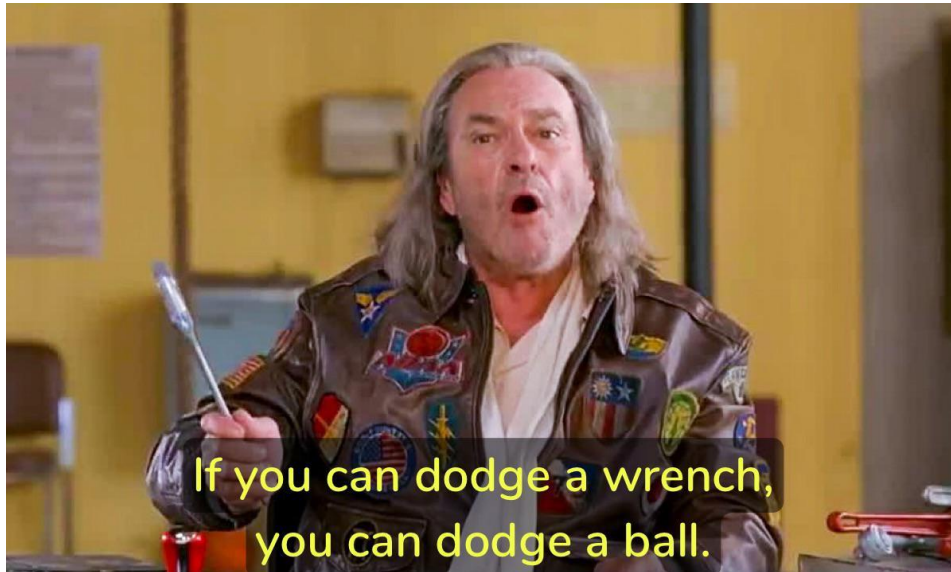
RAY (Core): A general-purpose distributed execution layer

Ray: A Unified System for ML



This is our focus for Lab 6

- If you can master Ray Tasks and Actors, learning the rest of the Ray libraries becomes much easier



The Ray API

Tasks

```
def f(shape):  
    return np.zeros(shape)
```

```
def add(a, b):  
    return a + b
```

The Ray API

Tasks

```
@ray.remote
```

```
def f(shape):  
    return np.zeros(shape)
```

```
@ray.remote
```

```
def add(a, b):  
    return a + b
```

The Ray API

Tasks

```
@ray.remote  
def f(shape):  
    return np.zeros(shape)
```

```
@ray.remote  
def add(a, b):  
    return a + b
```

```
o1 = f.remote([5, 5])
```

o1 is a:

...*future*: The eventual value will be computed by f.

...*remote reference*: The value may be stored on a remote node (in Ray's distributed object store).

The Ray API

Tasks

```
@ray.remote  
def f(shape):  
    return np.zeros(shape)
```

```
@ray.remote  
def add(a, b):  
    return a + b
```

```
o1 = f.remote([5, 5])
```

o1 is a:

...*future*: The eventual value will be computed by f.

...*remote reference*: The value may be stored on a remote node (in Ray's distributed object store).

The Ray API

Tasks

```
@ray.remote  
def f(shape):  
    return np.zeros(shape)
```

```
@ray.remote  
def add(a, b):  
    return a + b
```

```
o1 = f.remote([5, 5])  
o2 = f.remote([5, 5])  
o3 = add.remote(o1, o2)  
result = ray.get(o3)
```

o1 is a:

...*future*: The eventual value will be computed by `f`.

...*remote reference*: The value may be stored on a remote node (in Ray's distributed object store).

Demo!

The Ray API

Tasks

```
@ray.remote
```

```
def f(shape):  
    return np.zeros(shape)
```

```
@ray.remote
```

```
def add(a, b):  
    return a + b
```

```
o1 = f.remote([5, 5])  
o2 = f.remote([5, 5])  
o3 = add.remote(o1, o2)  
result = ray.get(o3)
```

```
class Counter(object):  
    def __init__(self):  
        self.value = 0  
    def inc(self):  
        self.value += 1  
        return self.value
```

The Ray API

Tasks

```
@ray.remote
def f(shape):
    return np.zeros(shape)
```

```
@ray.remote
def add(a, b):
    return a + b
```

```
o1 = f.remote([5, 5])
o2 = f.remote([5, 5])
o3 = add.remote(o1, o2)
result = ray.get(o3)
```

Actors

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
    return self.value
```

The Ray API

Tasks

```
@ray.remote
def f(shape):
    return np.zeros(shape)
```

```
@ray.remote
def add(a, b):
    return a + b
```

```
o1 = f.remote([5, 5])
o2 = f.remote([5, 5])
o3 = add.remote(o1, o2)
result = ray.get(o3)
```

Actors

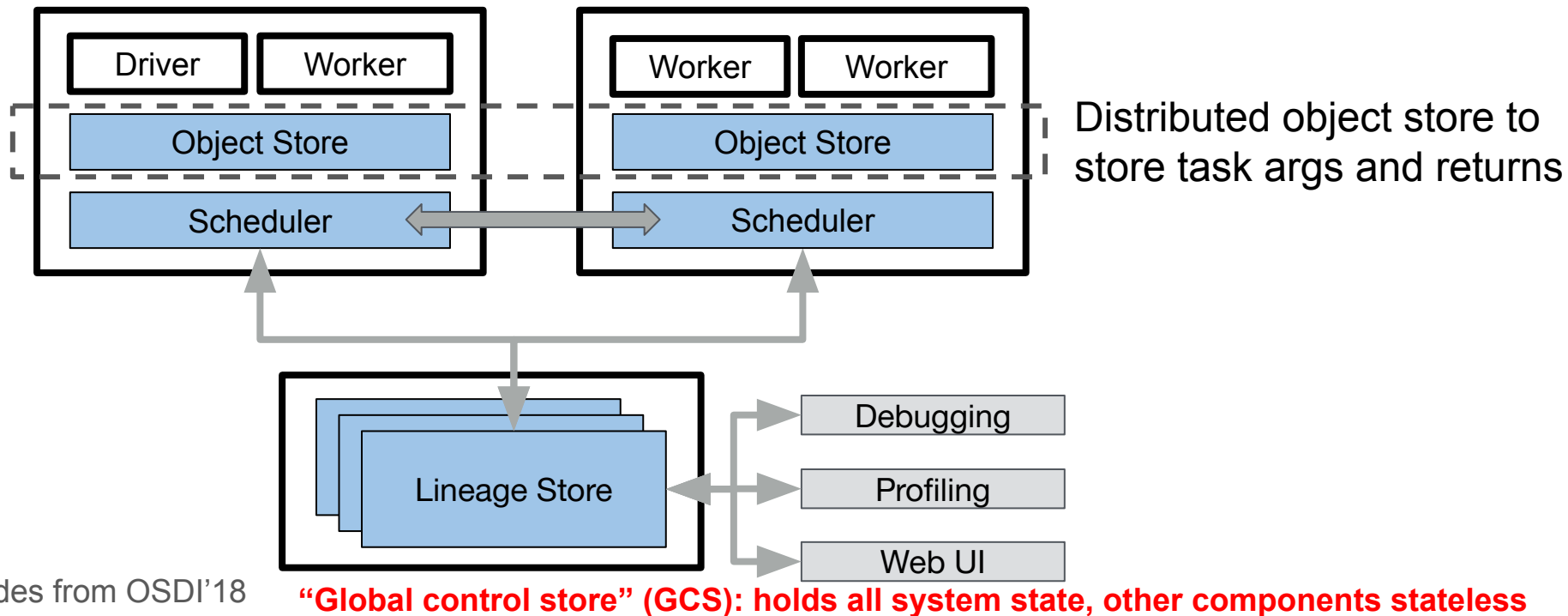
```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value
```

```
c = Counter.remote()
o4 = c.inc.remote()
o5 = c.inc.remote()
# Returns [1, 2].
result = ray.get([o4, o5])
```

Quick Break



2018: Ray pre-1.0 Architecture



Ray: Underneath the Hood

- **GCS** is designed to support scheduling millions of tasks / sec*
 - * = mileage may vary
- “Bottom-up scheduling” → first try to schedule tasks locally
 - Global scheduling only happens if/when node overloaded
- GCS is a (sharded and replicated) key-value store
 - Key: Object / Task IDs
 - Value: node location

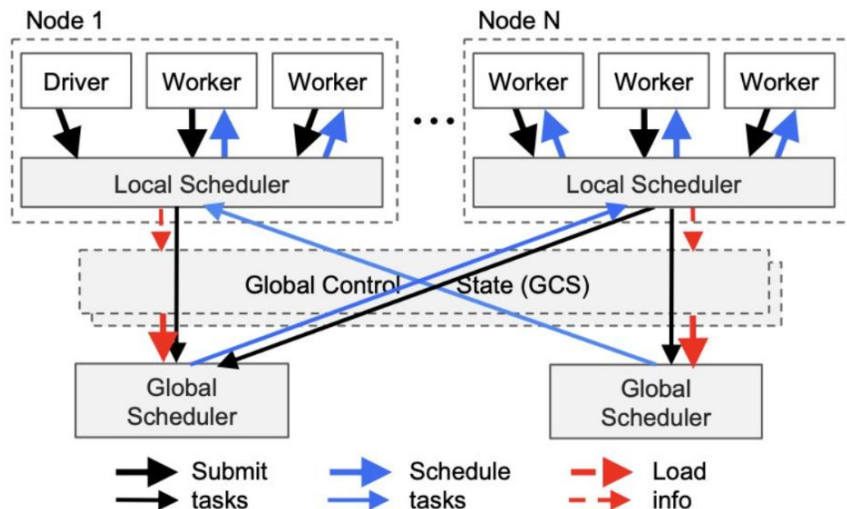
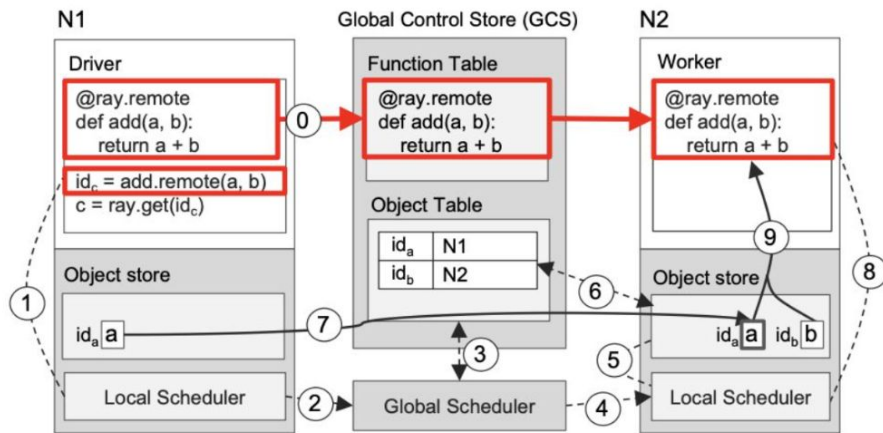


Figure 6: Bottom-up distributed scheduler. Tasks are submitted bottom-up, from drivers and workers to a local scheduler and forwarded to the global scheduler only if needed (Section 4.2.2). The thickness of each arrow is proportional to its request rate.

Example Task Execution

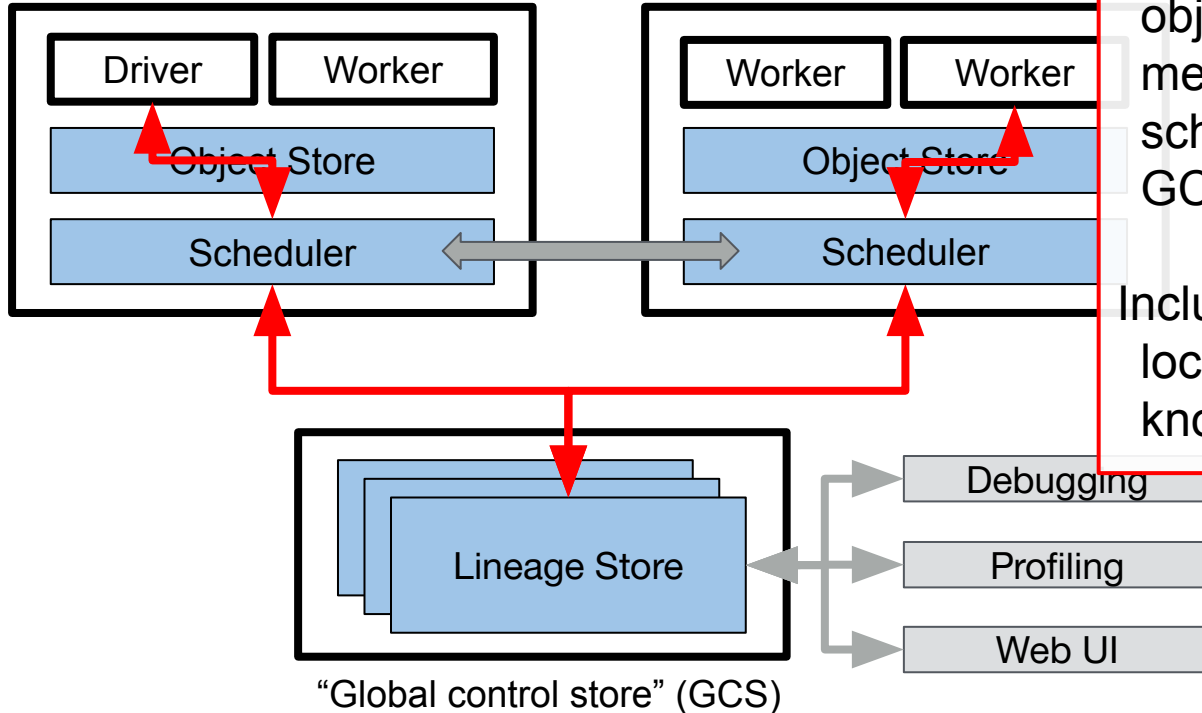
- Obj. A and B start on nodes N1 and N2 respectively
- Remote fcn. **add** is registered w/GCS upon init. and distributed to every worker (step 0)



(a) Executing a task remotely

- Steps (1-4): task submitted at N1 gets scheduled on N2 (for sake of ex.)
- Steps (5-7): input A is copied to N2 to bring all inputs to N2
- Steps (8-9): local sched. invokes task once inputs are ready

2018: Ray pre-1.0 Architecture

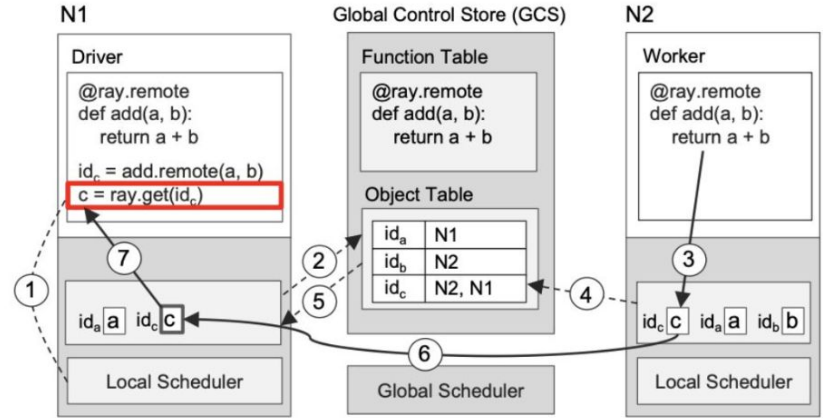


Performance: every task and object involved multiple messages with the scheduler, object store, and GCS.

Including actors (where location of worker is already known).

Example Returning Result of Execution

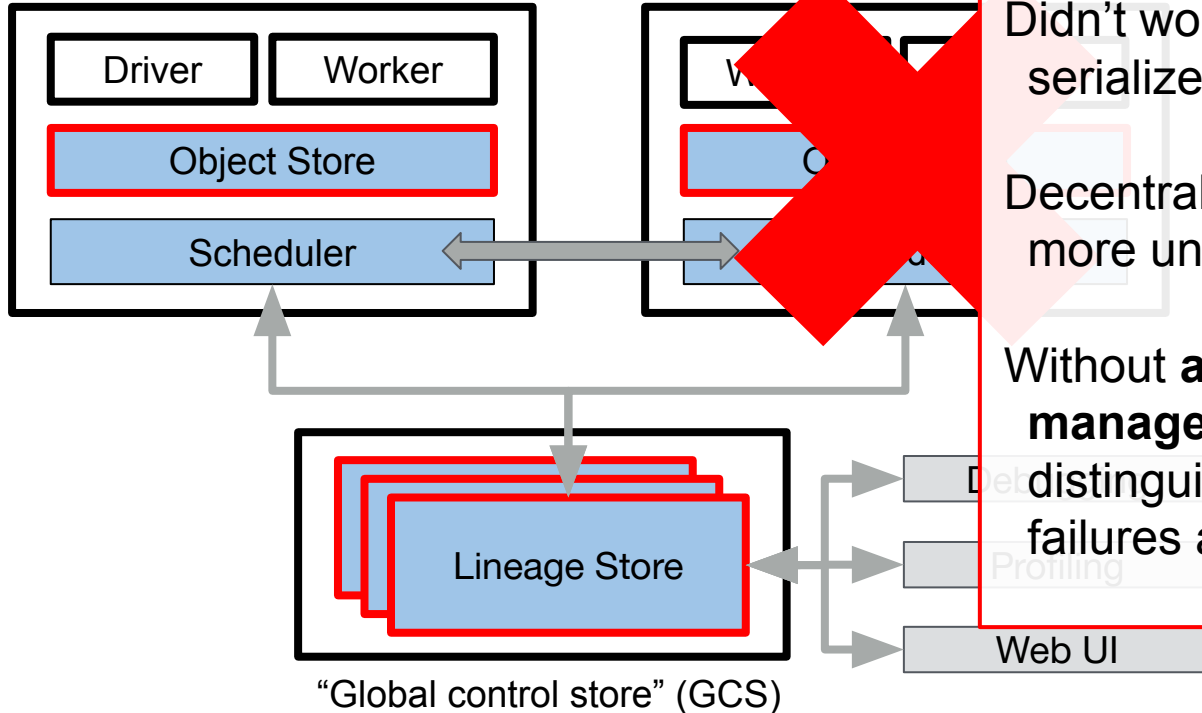
- Same setup as in previous slide
- Steps (1-2): lookup for value C results in N1's obj. store registering callback w/Object Table



(b) Returning the result of a remote task

- Steps (3-5): N2 completes execution of **add** and adds entry for C in Object Table
- Steps (5-7): callback is triggered; C is copied to N1 and returned

2018: Ray pre-1.0 Architecture



Fault tolerance:

Didn't work in a lot of cases: actors, serialized ObjectRefs, etc.

Decentralized design made system more unstable.

Without **automatic memory management**, could not distinguish between machine failures and OOM.

2018: Designing Ray v1.0

Problems:

- Decentralized design added a lot of **overhead**, especially for actor tasks.
 - System **complexity** created instability under load and failures.
 - Need automatic memory management for better stability.
 - But this would've added even more overhead and complexity!
- **Ray v1.0: We need to redesign the metadata control plane.**

2018: Designing Ray v1.0

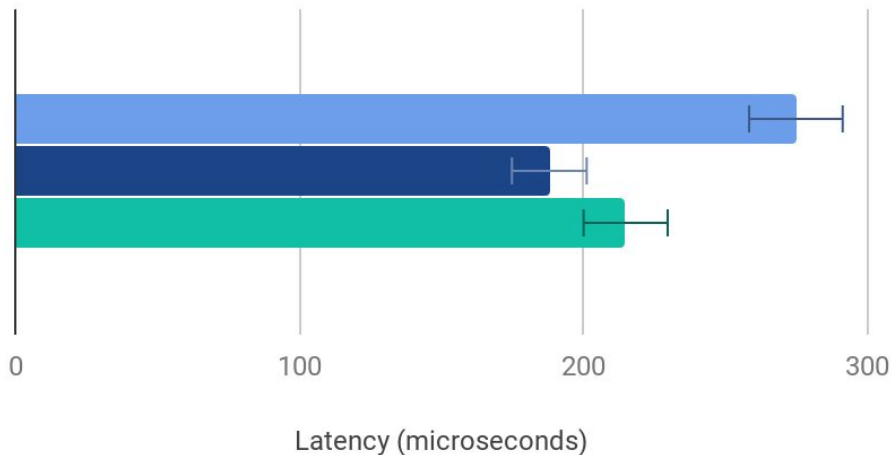
Some parallel Ideas:

- **Performance:** Reduce load from lower system components by having workers send tasks **directly** to each other via RPC.
- **Reducing complexity:** Instead of decentralizing by storing all system state in GCS, let's keep the decentralized part but introduce some notion of metadata **ownership**.
 - Who should the owner be? Automatic memory management makes this answer obvious: *the owner should be the original reference holder* (the worker that created the original ObjectRef)!

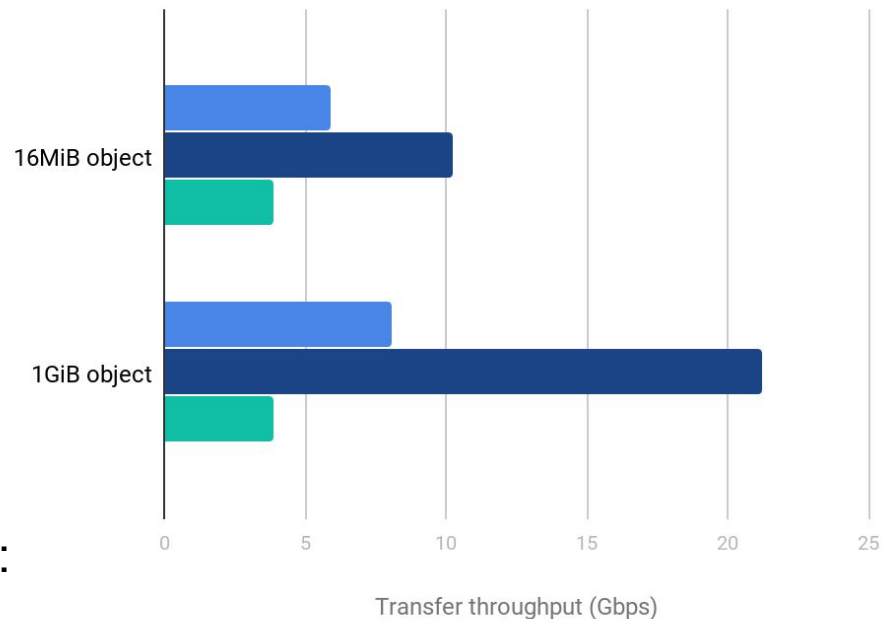
2018: Key metrics leading up to Ray v1.0



Actor Call Latency



Object Transfer Throughput



Stability changes that are harder to quantify:

- Task retries
- Automatic memory management

2020: A distributed futures system for fine-grained tasks

For generality, the system must impose low overhead.

Analogy: gRPC can execute millions of tasks/s. Can we do the same for **distributed futures**? → **futures whose values can be stored anywhere**

Goal: Build a distributed futures system that guarantees **fault tolerance with low task overhead**. → Note the similarity! :)

Enable applications that *dynamically* generate *fine-grained* tasks. → Check out the paper[1] for more details!

2020: Distributed futures introduce shared state

Legend



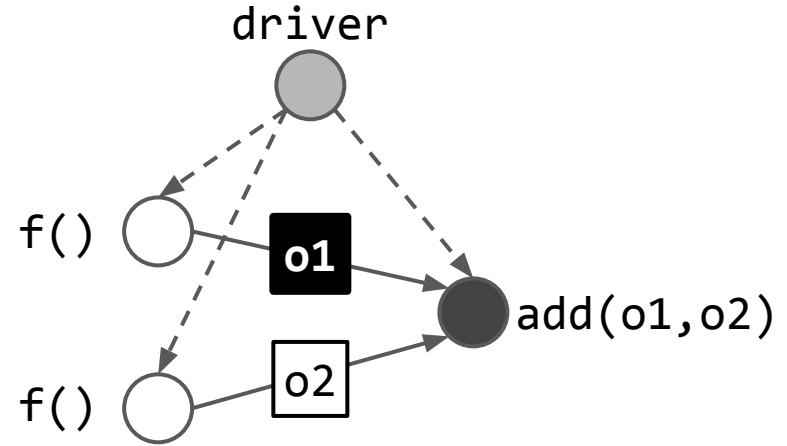
Task (RPC)



Invocation



Data dependency



Distributed Futures (in a nutshell)

a. RPC

- i. Function calls block; data is copied everywhere

b. RPC + distributed memory

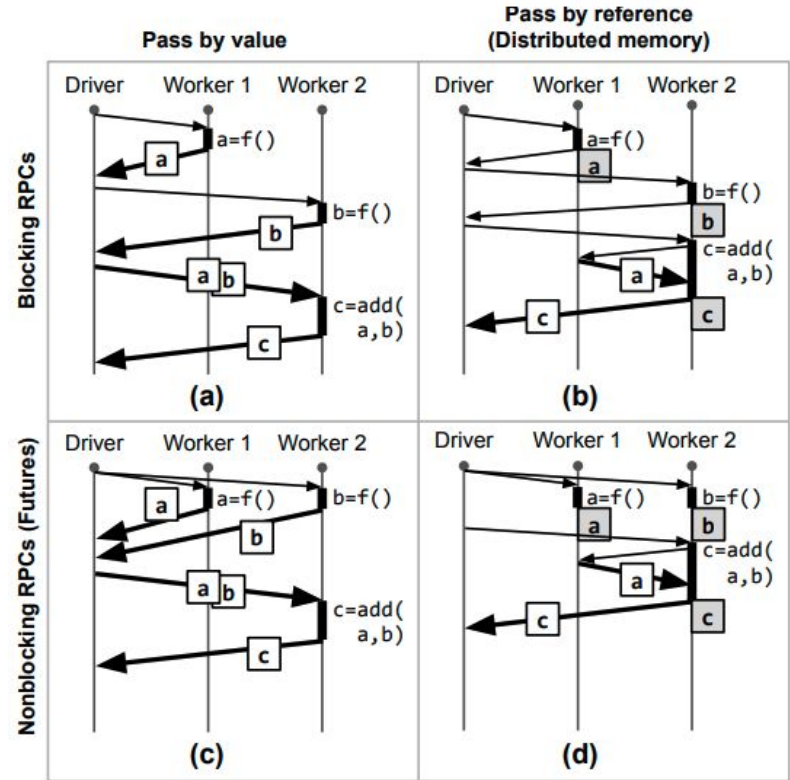
- i. Pass-by-reference eliminates some data copies

c. RPC + futures

- i. Functions can be executed in parallel

d. Distributed Futures

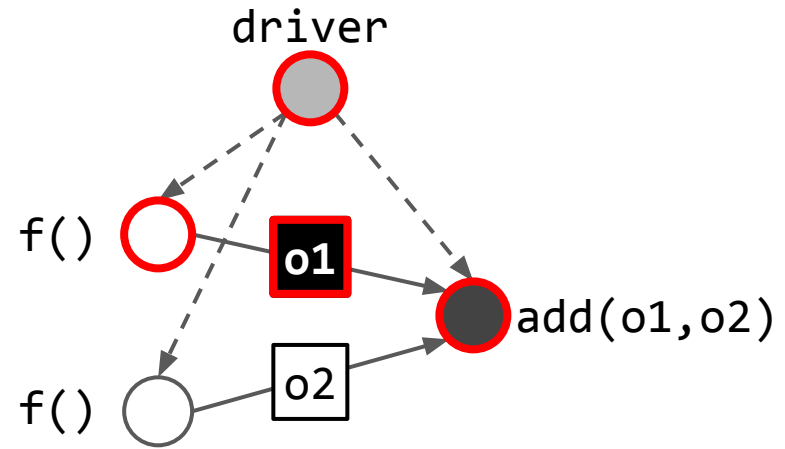
- i. Parallel execution & minimal data copy



2020: Distributed futures introduce shared state

Multiple processes refer to the same value.

1. The process that specifies how the value is created and used.
2. The process that creates the value.
3. The process that uses the value.
4. The physical location of the value.

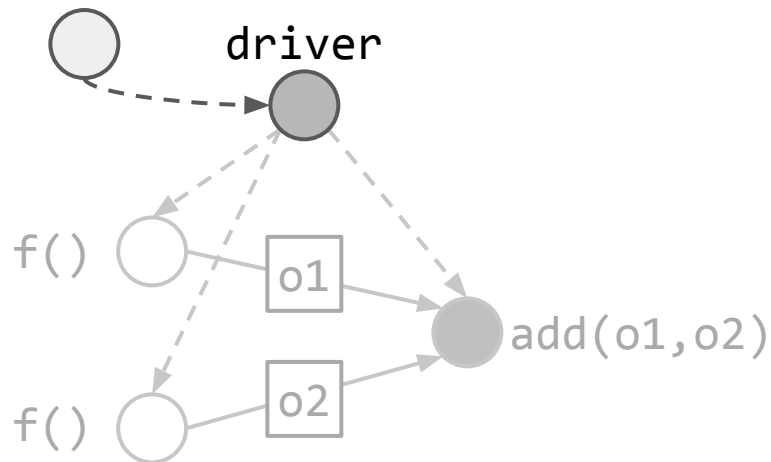


Dereferencing a distributed future requires **coordination**.

2020: Our approach: Ownership

Existing solutions do not take advantage of the inherent **structure** of a distributed futures application.

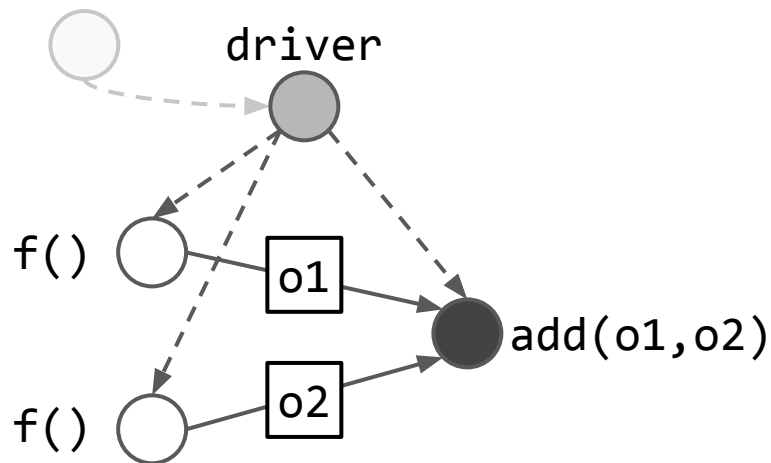
1. Task graphs are hierarchical.
2. A distributed future is often passed within the scope of the caller.



2020: Our approach: Ownership

Existing solutions do not take advantage of the inherent **structure** of a distributed futures application.

1. Task graphs are hierarchical.
2. A distributed future is often passed within the scope of the caller.








Insight: By leveraging the structure of distributed futures applications, we can decentralize without requiring expensive coordination.

2020: Our approach: Ownership

Insight: By leveraging the structure of distributed futures applications, we can decentralize without requiring expensive coordination.

Architecture	Failure handling	Performance
Ownership: The worker that calls a task <i>owns</i> the returned distributed future.	Each worker is a “ centralized owner ” for the objects that it owns. Use supervision to handle owner failure.	No additional writes on the critical path of task execution. Scaling through nested function calls .

Today: When to use Ray Core?

	Ray Tune/ Train	RLlib	Ray Serve	Ray Data	vLLM
Coarse-grained (process-level) orchestration					

Note: There are also benefits when composing libraries!

Ray Data: Scalable datasets for ML

Ray Data is a flexible and scalable data processing library

- + **Ease of use:** Python-native, easy deployment via Ray Core
- + **Transparent scale:** Transparent fault tolerance, resource management, data partitioning and placement, pipelining, heterogeneous clusters
- + **Flexibility:** Pipelining between **CPU and GPU tasks**; native support for tabular, image, (Anyscale-only) audio/video

Ray Data is a flexible and scalable data processing library

Offline use cases: Dataset creation

- Large-scale shuffle operations (deduping, groupby, etc)
- Batch inference
- Vector database and index creation

Online processing: Overlapping and scaling CPU+GPU applications

- Data loading + last-mile preprocessing for (distributed) training
- RAG pipelines

Ray Data is a flexible and scalable data processing library

Offline use cases: Dataset creation

- Large-scale shuffle operations (deduping, groupby, etc)
- Batch inference
- Vector database and index creation

Online processing: Overlapping and scaling CPU+GPU applications

- Data loading + last-mile preprocessing for (distributed) training
- RAG pipelines

Data loading for ML training

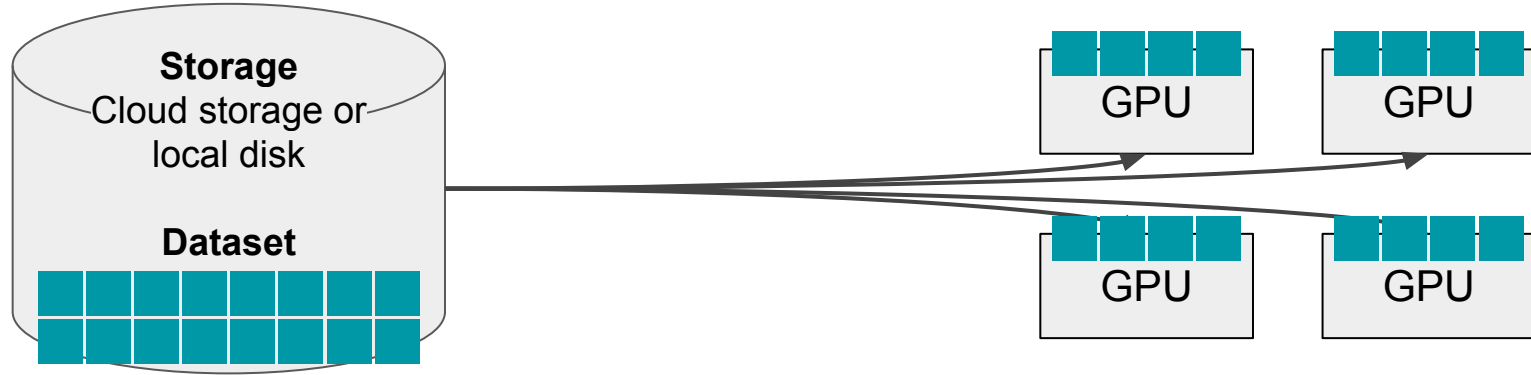


Needs to be **fast**, to maximize GPU utilization.

Needs to **scale** to large datasets and clusters.

→ Large dataset → Must stream through memory

Data loading for ML training



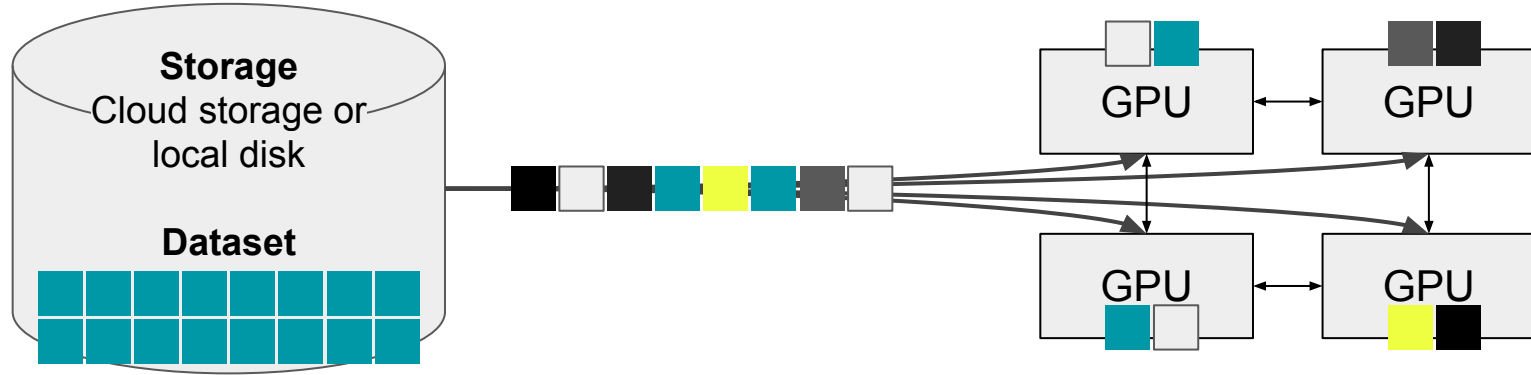
Needs to be **fast**, to maximize GPU utilization.

Needs to **scale** to large datasets and clusters.

→ Large dataset → Must stream through memory

→ Cluster → Must send data over the network

Data loading for ML training



Needs to be **fast**, to maximize GPU utilization.

Needs to **scale** to large datasets and clusters.

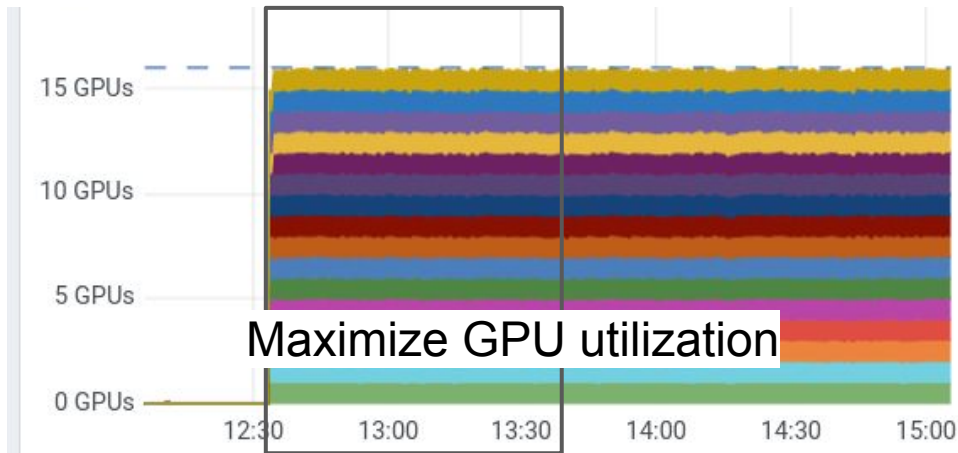
Needs to be **flexible**, to support arbitrary preprocessing.

→ Data can have different: storage, modality, preprocessing, memory footprint, ordering, ...

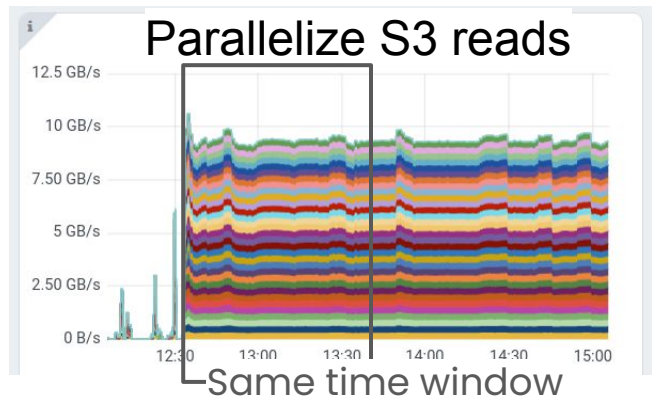
Ray Data is...

Training ResNet-50 (image classification) on a raw S3 dataset

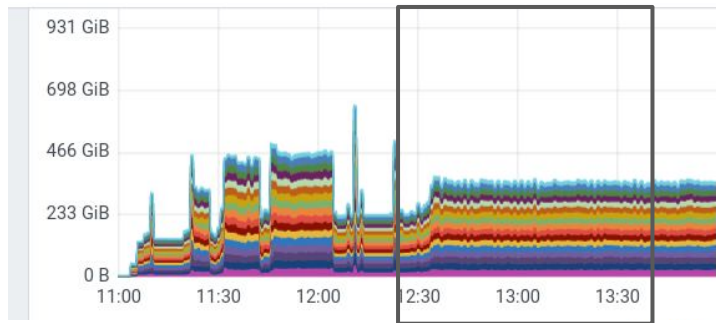
Fast



Streaming execution in v2.4+.
Shared-memory data loading.

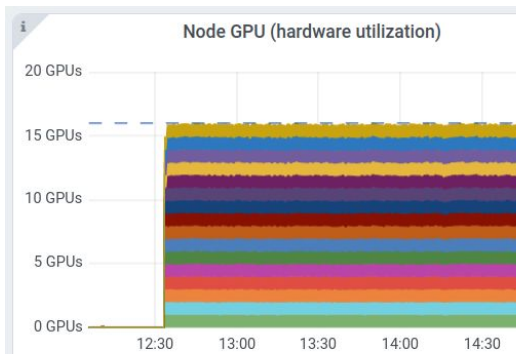


Control memory usage



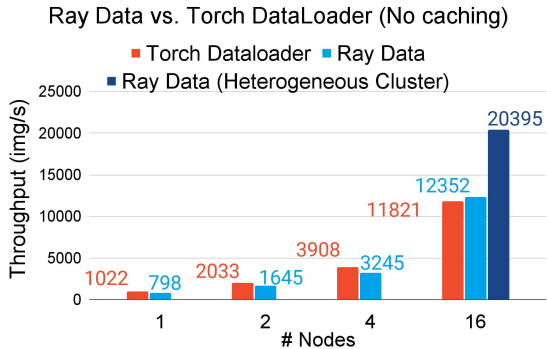
Ray Data is...

Fast



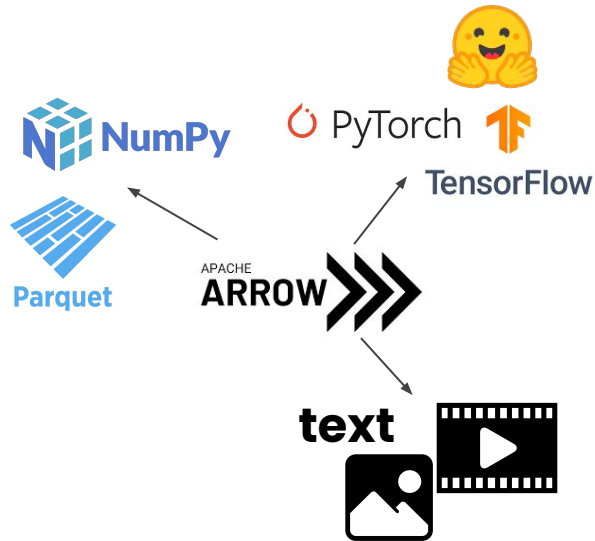
Streaming execution.
Shared-memory data loading.

Scalable



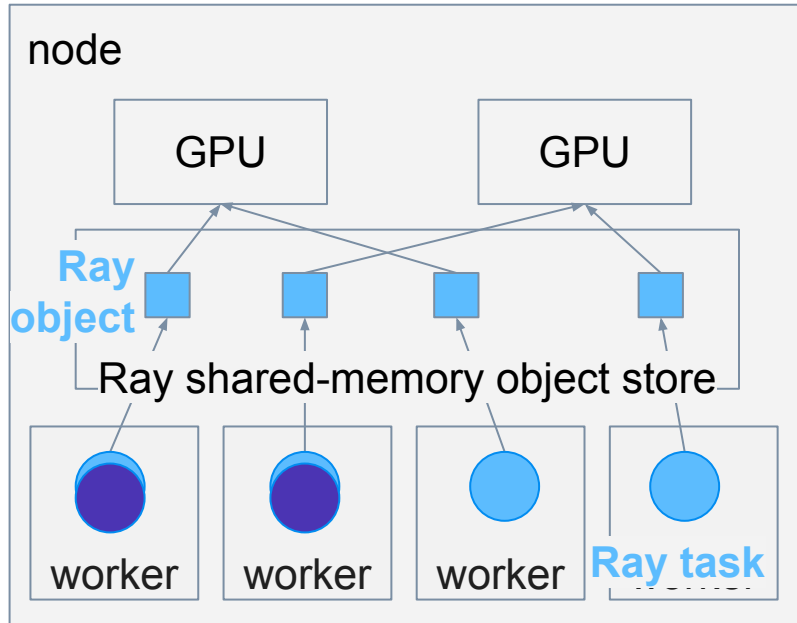
Heterogeneous clusters.
Automatic failure **recovery**.

Flexible



Query planner for building arbitrary data preprocessing pipelines.

Ray Data design



How are workers implemented?

→ Ray core → generic dist. compute

Ray Data design

Additional overheads compared to multiprocessing`:

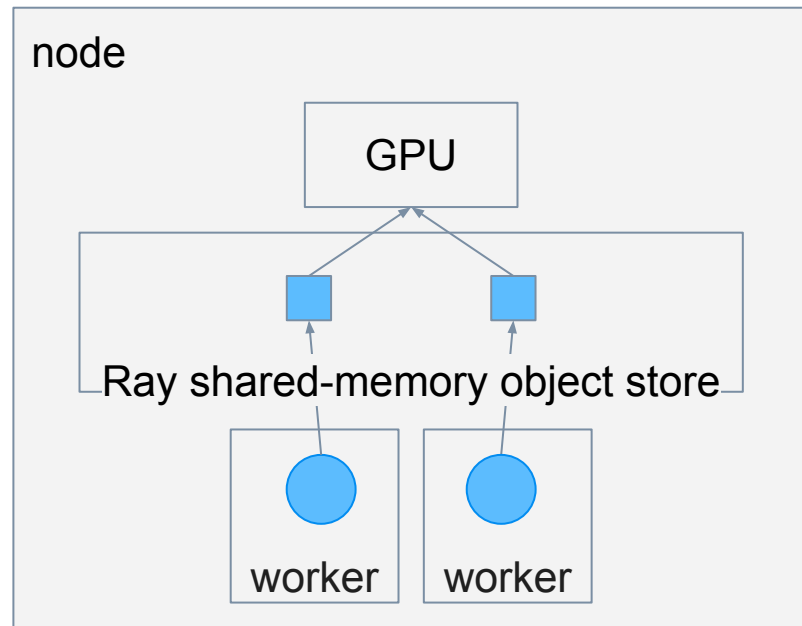
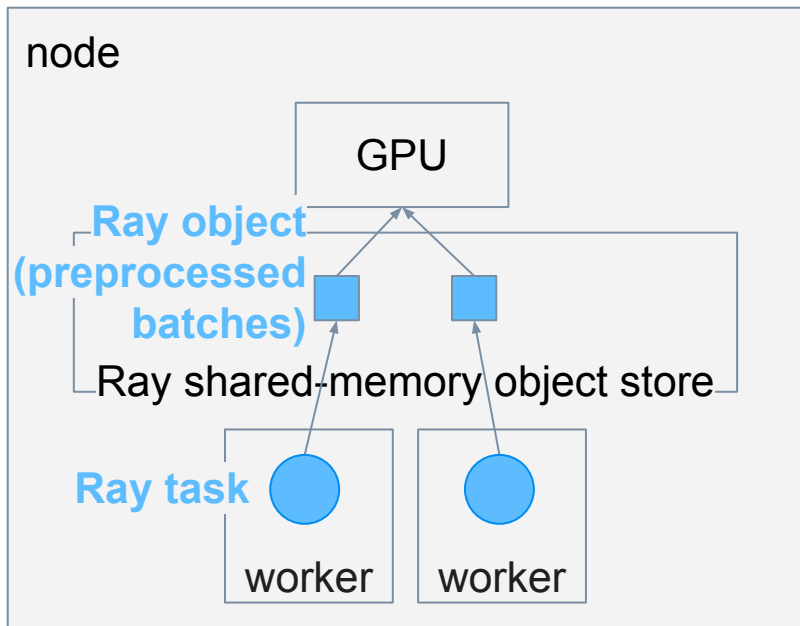
- Copy preprocessed data in shared memory
- Ray core task overhead (<1ms per task)

But in return:

- + **Automatically** partition data
- + Scheduler can control execution to dynamically **load-balance** and **limit memory usage**
- + **Recover from failures** without having to restart

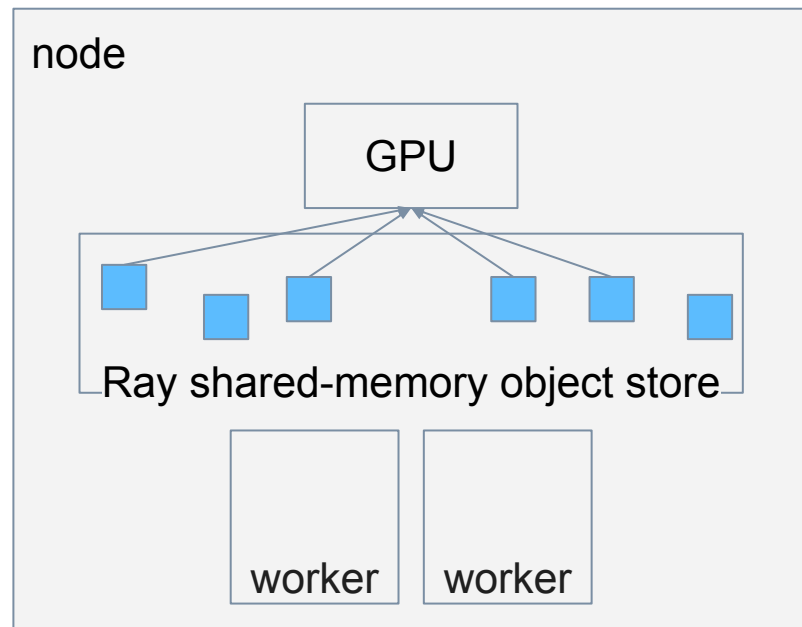
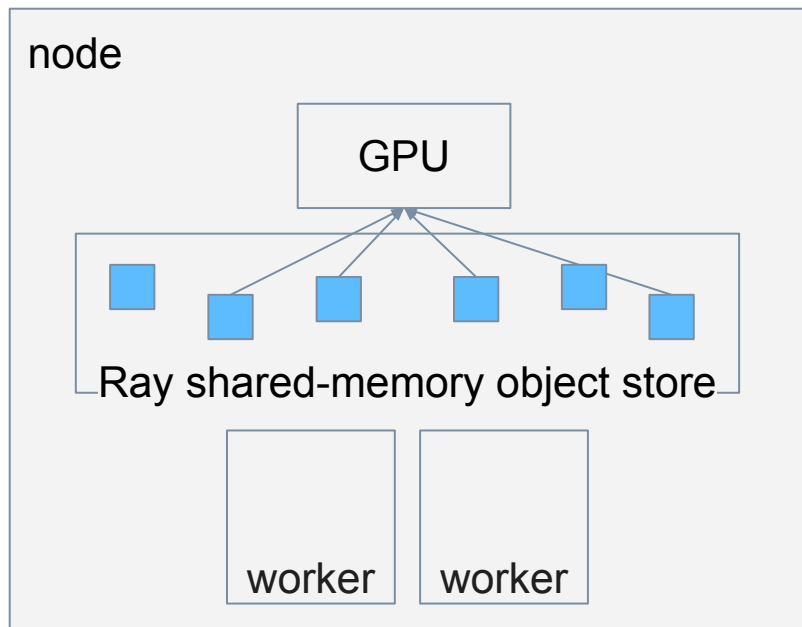
Get distributed features from Ray core “for free”.

Ray Data with **distributed** trainers



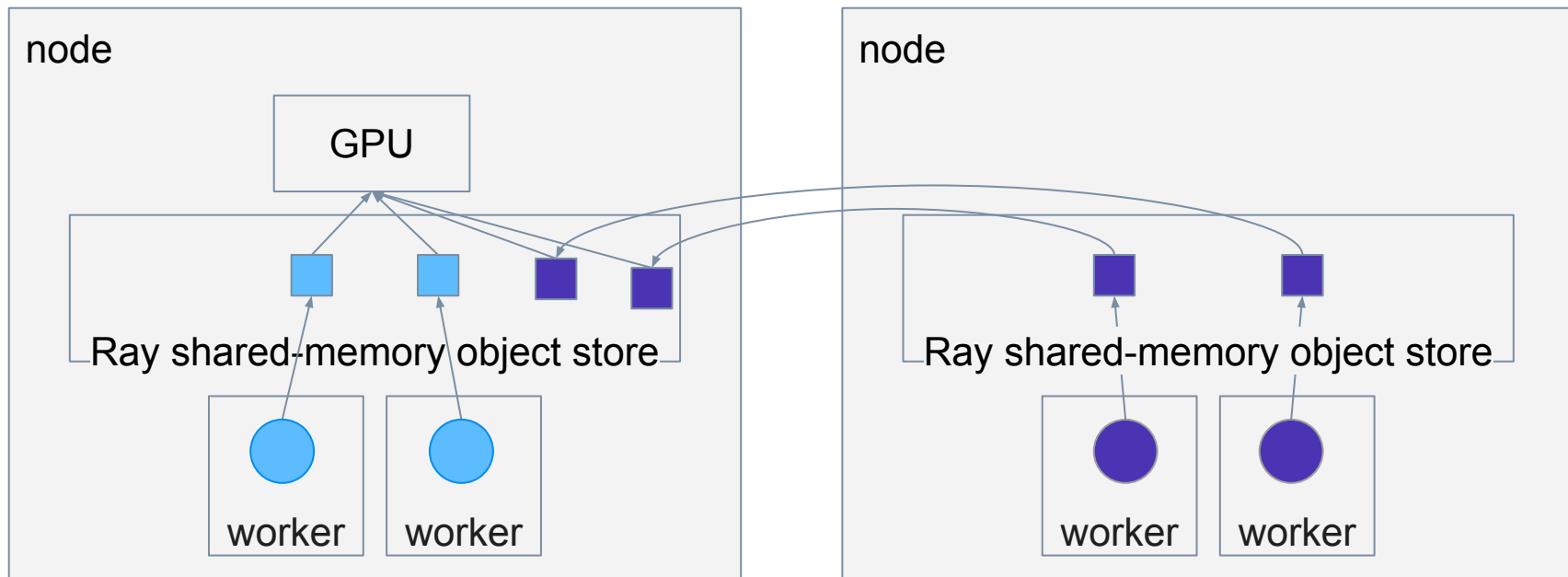
Ray Data routes batches based on **data locality** and **load-balancing**.

Caching Ray Datasets with `ds.materialize()`



Data can be cached at **any stage** of preprocessing. Ray core **automatically spills** to disk to avoid out-of-memory.

Ray Data with heterogeneous clusters



Data produced by remote tasks gets moved to the trainer node in the background.

Data loading for ML training features

Single-node + distributed:

- Automatic **dataset partitioning** and load-balancing across workers
- Automatic **memory limits**
- Recover from **failures without restarting** training
- **Cache** materialized datasets in-memory and on-disk

Distributed features:

- **Heterogeneous clusters**: Scale CPU-based data preprocessing separately from GPU-based training
- Locality-based scheduling
- (soon) Autoscaling clusters

Flexibility!

ImageNet scalability benchmark on S3



1. Load images from S3 path

```
ds = ray.data.read_images(  
    "s3://bucket"  
)  
ds = ds.map(  
    crop_and_flip_image  
)
```


ImageNet scalability benchmark on S3



1. Load images from S3 path
2. **Apply preprocessing fn to images**

```
ds = ray.data.read_images(  
    "s3://bucket"  
)  
ds = ds.map(  
    crop_and_flip_image  
)
```

Query
optimizer

```
ds = ds.map(  
    read_images->  
    crop_and_flip_image  
)
```

ImageNet scalability benchmark on S3



1. Load images
from S3 path

```
ds = ray.data.read_images(  
    "s3://bucket"  
)  
ds = ds.map(  
    crop_and_flip_image  
)
```

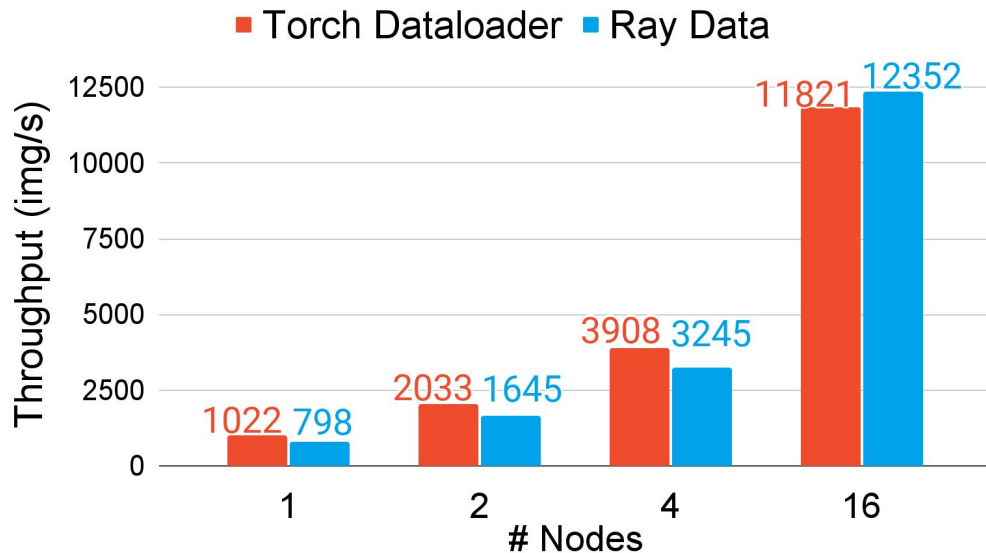
2. Apply
preprocessing
fn to images

3. **Ingest with
Ray Train
TorchTrainer**

```
def train_loop_per_worker(batch_iter : ray.data.DataIterator):  
    # Batch Iterator over Ray Dataset/Torch DataLoader  
    for batch in batch_iter:  
        ...
```

ImageNet scalability benchmark on S3

Ray Data vs. Torch DataLoader (No caching)



Node setup: g4dn.xlarge

- 16 vCPU
- 1 NVIDIA T4 GPU
- 64 GiB memory

Dataset:

- ImageNet, stored as raw images (JPG) on S3
- Each trainer reads about 10GB of images

Ray Data is **fast** and **scalable**, matching manually tuned Torch DataLoader in a distributed setting.

Implementation Details



```
ds = ray.data.read_images(  
    "s3://bucket"  
)  
ds =  
ds.map(crop_and_flip_image)
```

Torch required tedious wrangling:

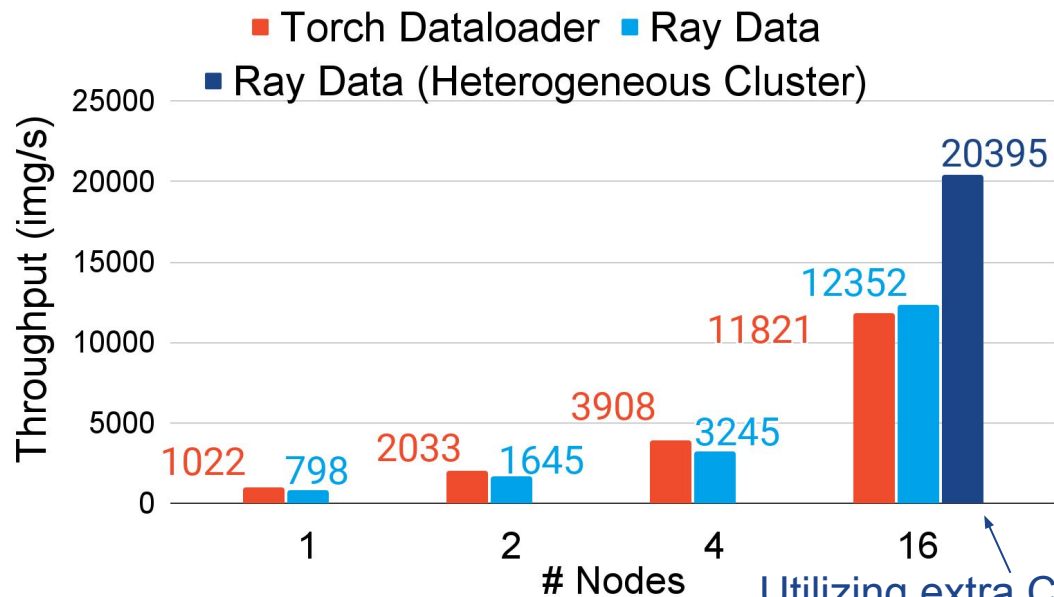
- Custom S3 DataLoader implementation
- Manual tuning of num_workers
- Manual partitioning of input dataset files to each worker



```
def load_image(inputs):  
    import io  
    from PIL import Image  
  
    url, fd = inputs  
    data = fd.file_obj.read()  
    image = Image.open(io.BytesIO(data))  
    image = image.convert("RGB")  
    if transform is not None:  
        image = crop_and_flip_image(image)  
    return image  
  
class FileURLDataset:  
    def __init__(self, file_urls):  
        self._file_urls = file_urls  
  
        def __iter__(self):  
            worker_info =  
torch.utils.data.get_worker_info()  
            assert worker_info is not None  
  
            torch_worker_id = worker_info.id  
            return  
        iter(self._file_urls[torch_worker_id])  
  
        file_urls = INPUT_FILES_PER_WORKER[worker_rank]  
        file_urls = [f.tolist() for f in  
np.array_split(file_urls, num_workers)]  
        file_url_dp =  
IterableWrapper(FileURLDataset(file_urls))  
        file_dp = S3FileLoader(file_url_dp)  
        image_dp = file_dp.map(load_image)
```

Supporting Heterogeneous Clusters

Ray Data vs. Torch DataLoader (No caching)



Node setup: g4dn.xlarge
+ 4 r5.16xlarge

- 16 vCPU
- 1 NVIDIA T4 GPU
- 64 GiB memory
- +64 vCPU, 512 GiB memory

Dataset:

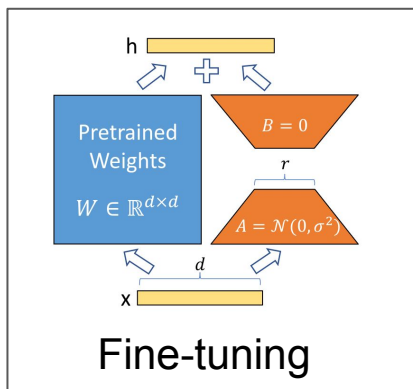
- ImageNet, stored as raw images (JPG) on S3
- Each trainer reads about 10GB of images

Utilizing extra CPUs to
maximize throughput
No code changes!

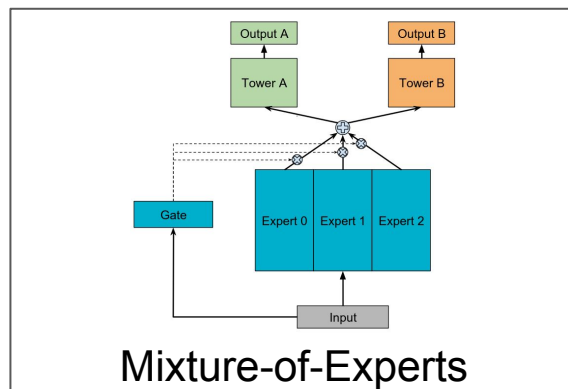
What's next for Ray?

Trend: Model execution is becoming more complex

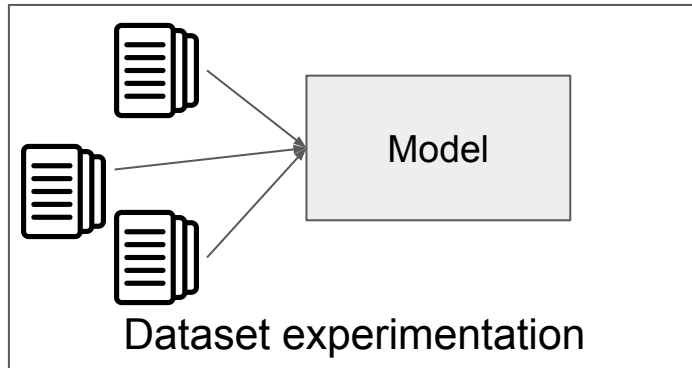
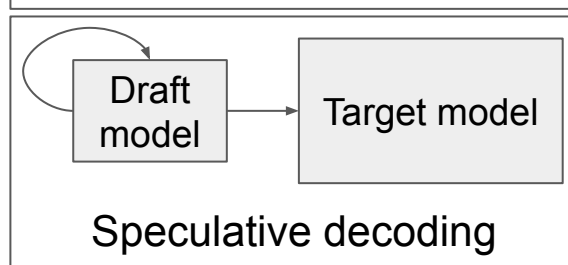
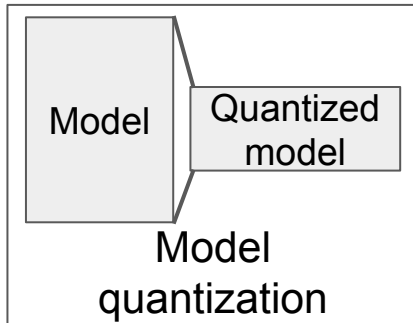
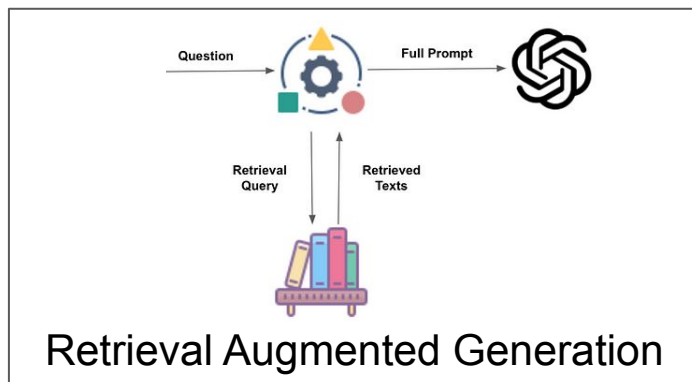
Post-training



Model routing



Data processing



Trend: Model execution is becoming more complex

Simple scaling of models is getting increasingly expensive.

Before: **One model per task**, all inputs take the **same path**.

After: **One to many models for many tasks**, inputs may take **different and dynamic paths**.

Meanwhile, current (distributed) ML systems are highly **static!**

Is Ray Core the answer?

	Ray Tune/ Train	RLlib	Ray Serve	Ray Data	vLLM
Coarse-grained (process-level) orchestration					
Fine-grained (10ms+ function-level) orchestration					
Distributed memory management					

Is Ray Core the answer?

	Ray Tune/ Train	RLlib	Ray Serve	Ray Data	vLLM
Coarse-grained (process-level) orchestration	✓	✓	✓	✓	✓
Fine-grained (10ms+ function-level) orchestration		✓	✓	✓	✓
Distributed memory management				✓	

Problem: GPU “tasks” run at 100s of us.

Is Ray Core the answer?

	Ray Tune/ Train	RLlib	Ray Serve	Ray Data	vLLM
Coarse-grained (process-level) orchestration	✓	✓	✓	✓	✓
Fine-grained (10ms+ function-level) orchestration		✓	✓	✓	✓
Distributed memory management				✓	

Problem: GPU memory management is often tightly coupled with GPU compute.

Ray 3.0: Accelerated DAGs

Observation 1: Ray Core is (relatively) slow because it assumes a completely dynamic workload.

Observation 2: Even complicated GPU schedules like pipeline parallelism are not very dynamic.

Ray 3.0: Accelerated DAGs

Key ideas:

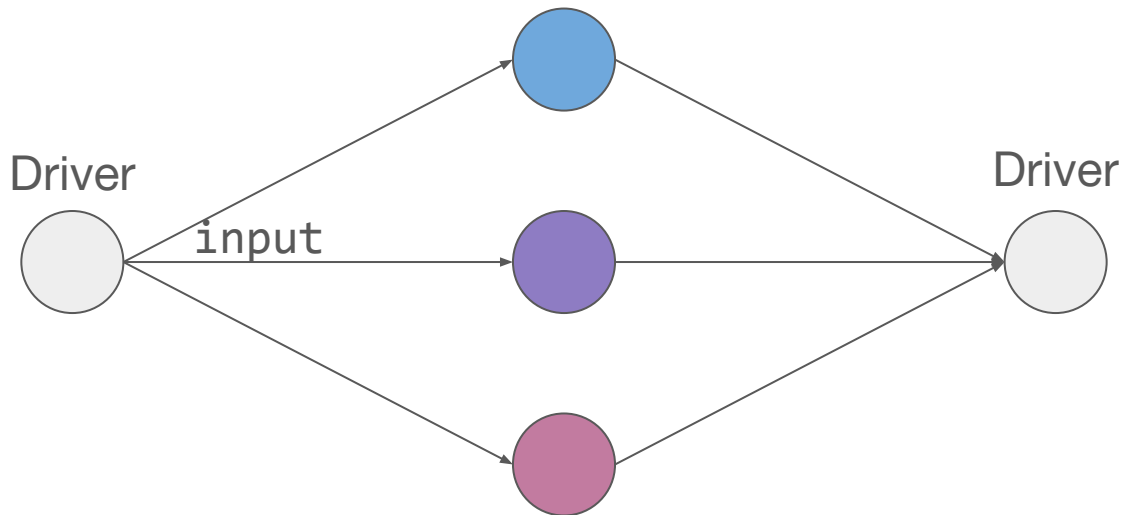
- (Initially) Restrict user to static dataflow
- Provide fast, transparent, pipelined data movement between GPUs

Goal: Reduce burden in building (distributed) GPU systems, without loss of performance.

Ray 3.0: Accelerated DAGs

Tensor-parallel inference DAG:

```
dag = ray.dag.MultiOutputNode(  
    [w.fwd.bind(input) for w in workers])
```



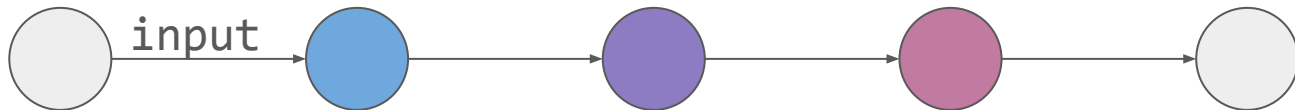
Ray 3.0: Accelerated DAGs

Pipeline-parallel DAG:

```
dag = input
```

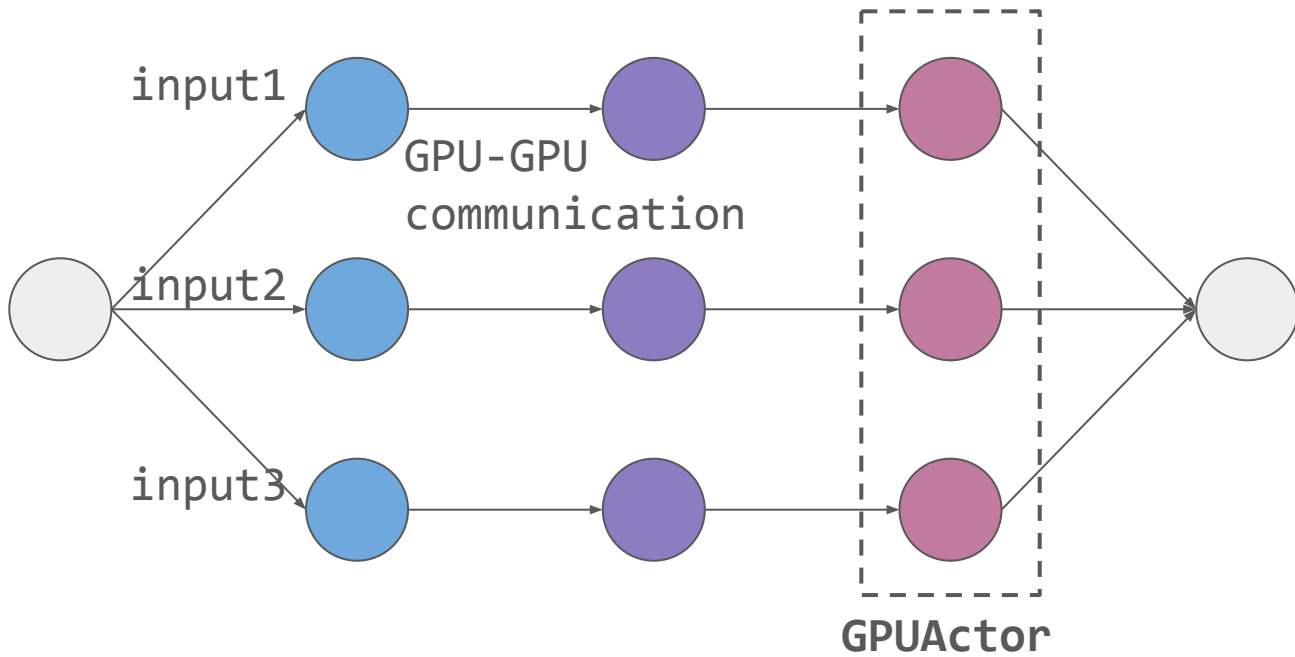
```
for w in workers:
```

```
    dag = w.fwd.bind(dag)
```



Ray 3.0: Accelerated DAGs

Pipeline-parallel DAG:



Ray 3.0: Accelerated DAGs for LLM inference

Current use cases:

- Prefill disaggregation
- Pipeline parallelism

Experimental use cases:

- Mixing tensor parallelism and pipeline parallelism
- CPU offloading
- Heterogeneous GPU systems
- Online prompt processing