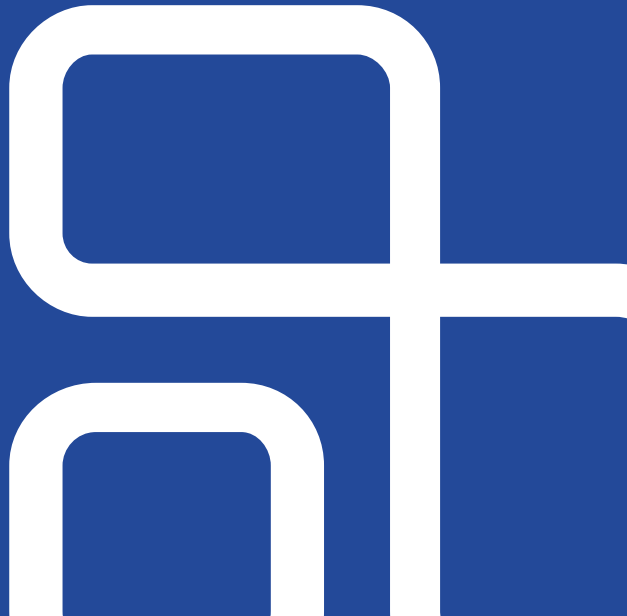
Ray for Large-Scale Data Processing

Clark Zinzow
Software Engineer @ Anyscale



anyscale

What's this talk about?

An overview of Ray and its ecosystem

Data processing on Ray, with a focus on Dask-on-Ray

A bit about me

Software engineer at Anyscale

Working on the core Ray system, focusing on supporting large-scale data processing

Before Anyscale, I was a Ray user!

What is Ray?

And why should you care?

What is Ray?

A distributed task execution engine providing a simple API for building distributed applications.

Core system design goals are **performance** and **reliability**, leveraging smart decentralized scheduling, fault-tolerant protocols, and a high-performance distributed in-memory object store to achieve high task throughput in the face of failures.

Python-centric, although we have Java and C++ APIs!

Ecosystem is currently ML-focused, although we're continuously broadening that scope.

Ray API and Programming Model

anyscale

Function

```
def read_array(file):
    # read ndarray “a”
    # from “file”
    return a


def add(a, b):
    return np.add(a, b)

a = read_array(file1)
b = read_array(file2)
sum = add(a, b)
```

Class

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

c = Counter()
c.inc()
c.inc()
```

anyscale

Function → Task

```
@ray.remote
def read_array(file):
    # read ndarray “a”
    # from “file”
    return a

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

a = read_array(file1)
b = read_array(file2)
sum = add(a, b)
```

Class → Actor

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

c = Counter()
c.inc()
c.inc()
```

anyscale

Function → Task

```
@ray.remote
def read_array(file):
    # read ndarray “a”
    # from “file”
    return a

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

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

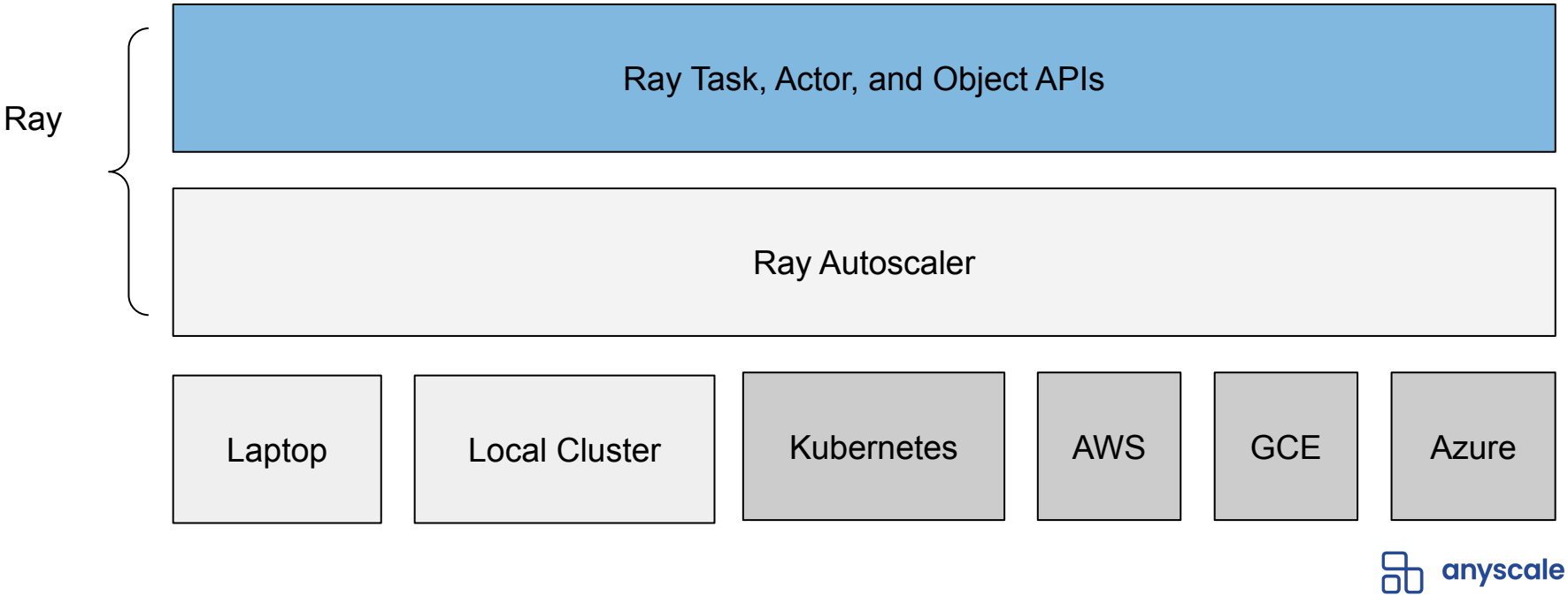
Class → Actor

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

c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
```

anyscale

Ray API Principles



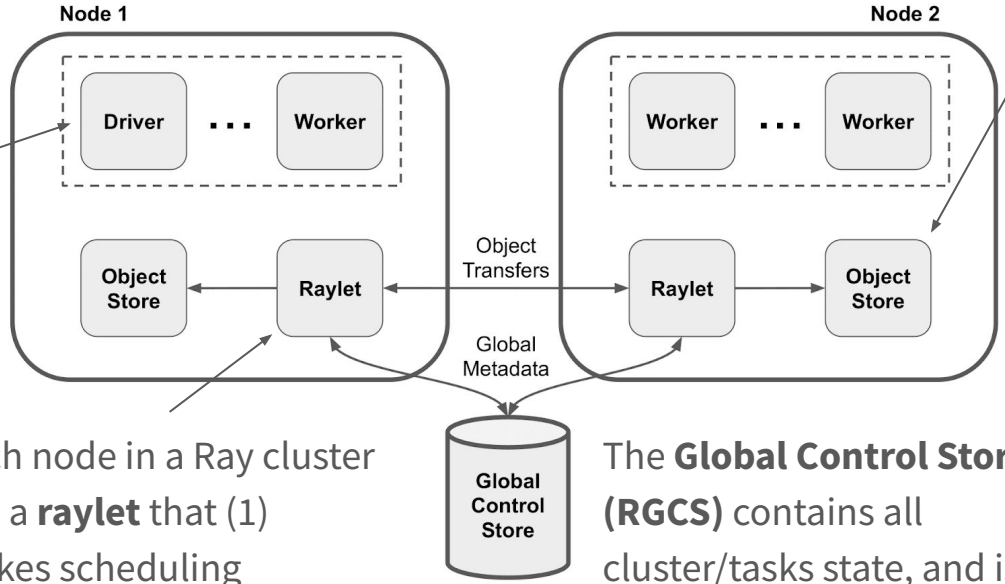
anyscale

Ray Architecture

anyscale

Ray Architecture Overview

A **driver** is a Ray client (application code) and is colocated with one or more Ray **workers**



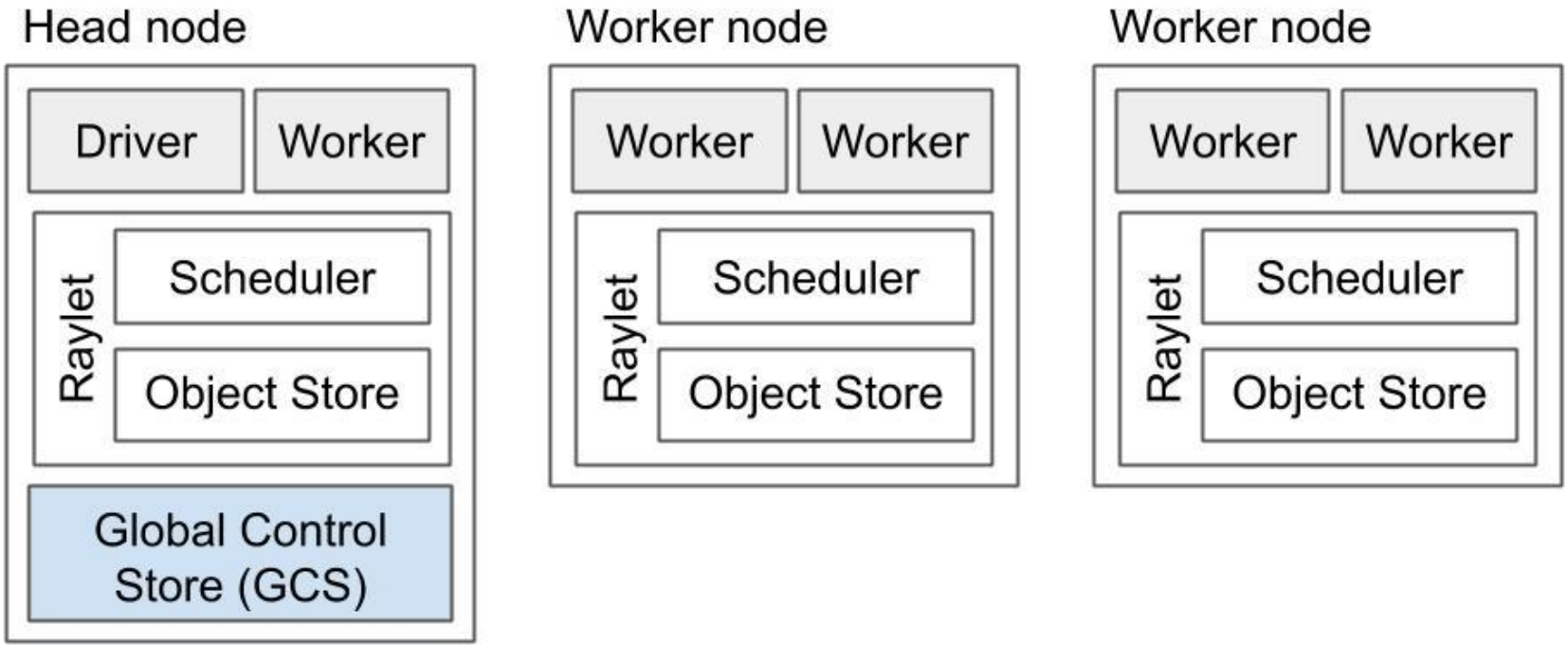
The **object store (Plasma)** is used for persisting and communicating all task/actor inputs and outputs; workers only read/write from/to their local object store, with the **raylet** facilitating transfers to other workers

Each node in a Ray cluster has a **raylet** that (1) makes scheduling decisions for tasks created by local drivers/workers, and (2) facilitates object transfers between nodes

The **Global Control Store (RGCS)** contains all cluster/tasks state, and is the only truly stateful component of a Ray cluster

anyscale

A Ray Cluster



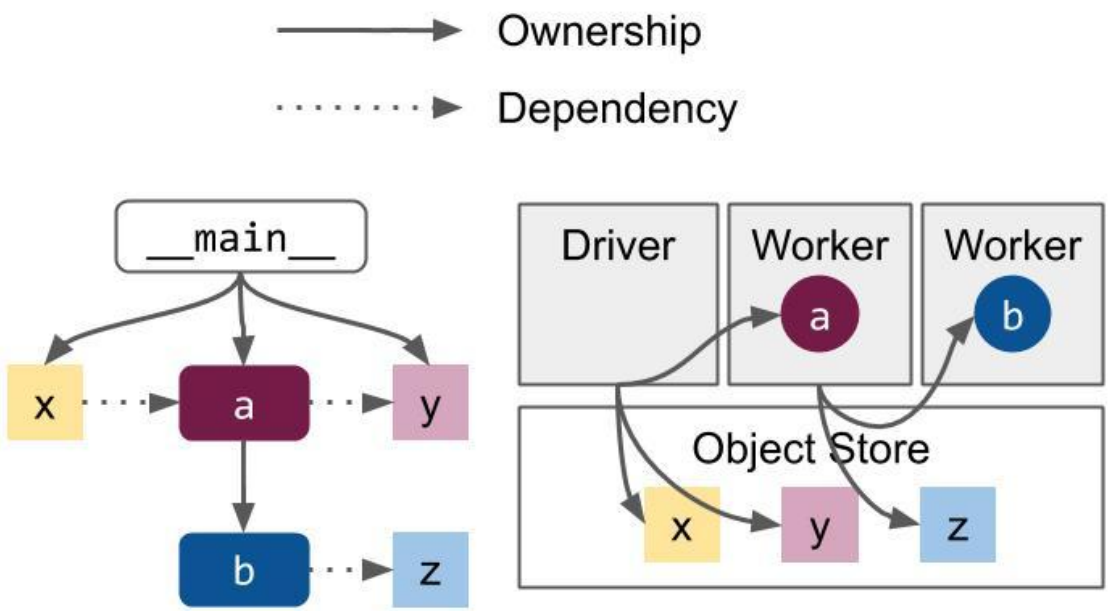
anyscale

Under the Hood

```
@ray.remote
def b():
  return

@ray.remote
def a(dep):
  z = b.remote()

x = ray.put(...)
y = a.remote(x)
```



Program **Task graph** **Physical execution**

Comparison to Other Systems

Stateful computation

Actor fault tolerance

Performance

- Millions of tasks per second, millisecond latencies
- Shared memory, zero-copy serialization

Built on gRPC and Arrow, C++ cross language bindings

- Supports Python, C++, and Java (cross-language applications)

Why is Ray great for data processing?

System Design

Ecosystem

Ray System Design

Heterogeneous resources

Distributed in-memory object store

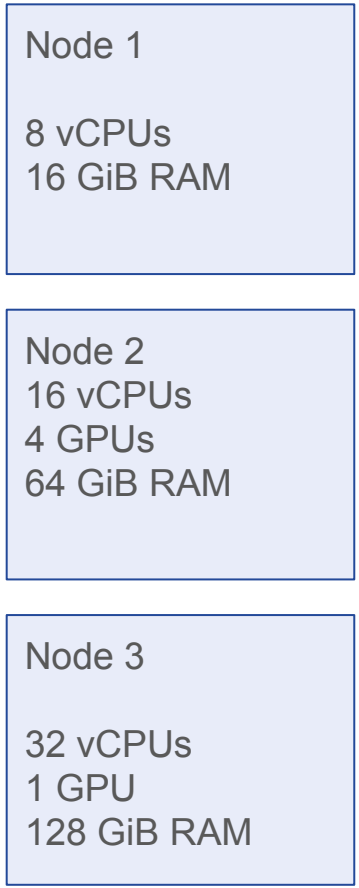
Fast, smart, decentralized scheduling

Heterogeneous Resources

Ray can house disparate node types on a single cluster:

- Differing hardware (CPUs and GPUs)
- Differing number of CPU cores
- Differing amount of CPU memory

These heterogeneous resources are all schedulable at the task and actor level, enabling fine-grained resource provisioning



Distributed In-memory Object Store

Performance:

Intermediate results transparently cached in memory, transferred over network

Zero-copy reads on same-node workers via shared memory

Reliability:

Ray scheduler limits how much total memory can be used by objects on single node

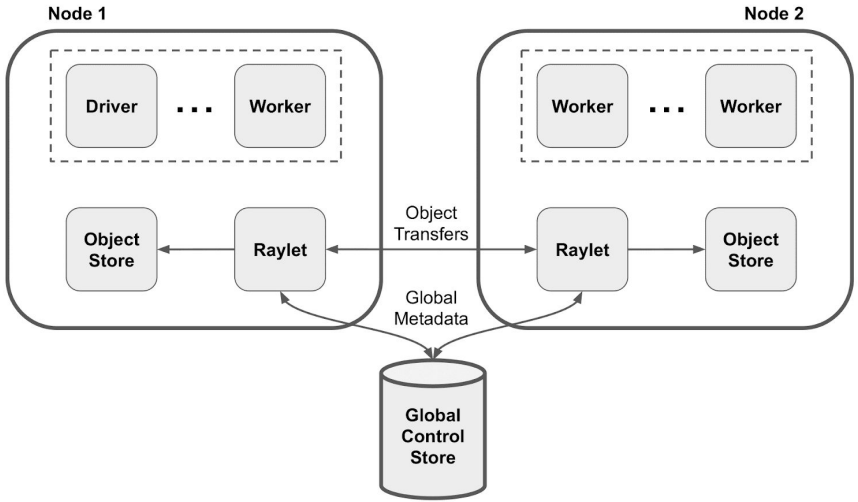
When object store is full, objects are spilled to external storage

Scheduling

Decentralized scheduling => massive scalability

Hot node mitigation - will attempt to schedule tasks onto nodes with low memory utilization

Locality-aware scheduling - will try to schedule tasks on node with most task dependency bytes already local



anyscale

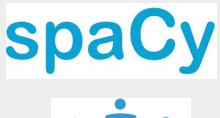
Ray Ecosystem

Data processing, model training, model serving, reinforcement learning, hyperparameter tuning

Ray Ecosystem

Native Libraries

3rd Party Libraries



anyscale

Integration in Focus: Dask-on-Ray

anyscale

What is Dask?

anyscale

What is Dask?

Dask is a parallel computing library for Python, geared towards scalable analytics

Dask provides parallel ndarray and dataframe abstractions that extend the interfaces of NumPy and Pandas to larger-than-memory workloads and distributed environments

Dask collections and ecosystem

Dask arrays mimic the NumPy API

```
# Arrays implement the Numpy API
import dask.array as da
x = da.random.random(size=(10000, 10000),
                     chunks=(1000, 1000))
x + x.T - x.mean(axis=0)
```

Dask dataframes mimic the Pandas API

```
# Dataframes implement the Pandas API
import dask.dataframe as dd
df = dd.read_csv('s3://.../2018-*-*.csv')
df.groupby(df.account_id).balance.sum()
```

Dask-ML implements the sklearn API, allowing for model training and inference to run on Dask

```
# Dask-ML implements the Scikit-Learn API
from dask_ml.linear_model \
  import LogisticRegression
lr = LogisticRegression()
lr.fit(train, test)
```

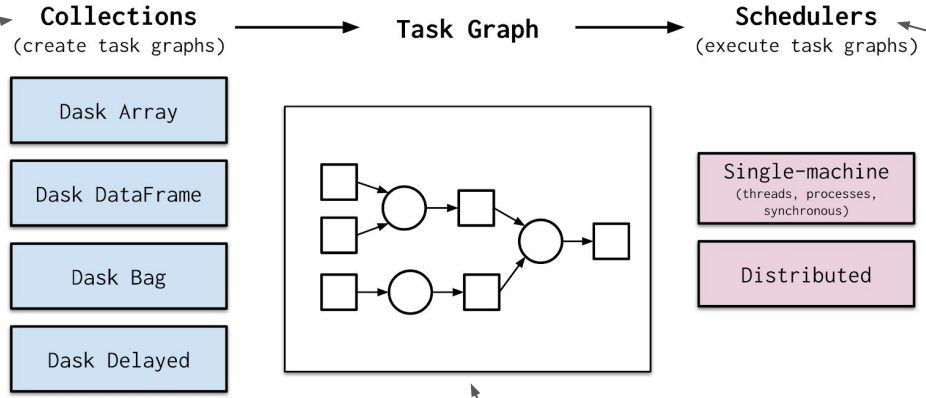
Dask bags mimic the iterators, Toolz, and PySpark APIs

```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(
    10, lambda pair: pair[1]).compute()
```

anyscale

Dask's three core components

The collection abstractions, mimicking the interfaces of familiar data science libraries



Task schedulers and execution backends, which schedule tasks for execution on a cluster of cores and/or machines

The task graph specification, representing collection operations as a DAG of function application tasks

anyscale

Dask task graph specification

The Dask task graph captures the data dependencies of tasks

Each node represents an operation (function) and each edge represents data dependencies between two operations (return values and arguments)

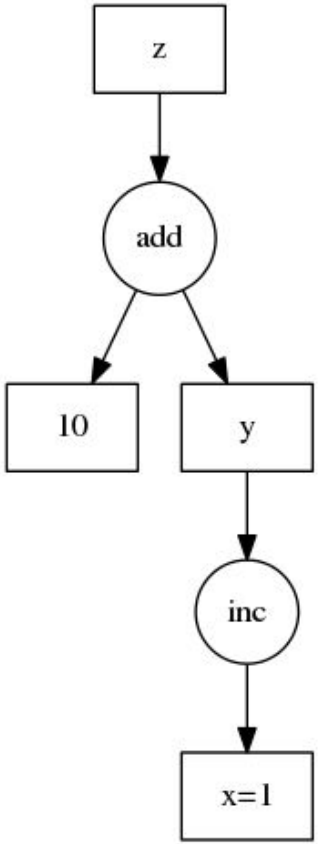
The Dask task graph is explicitly represented in memory, as a Python dictionary

```
def inc(i):
    return i + 1

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

x = 1
y = inc(x)
z = add(y, 10)
```

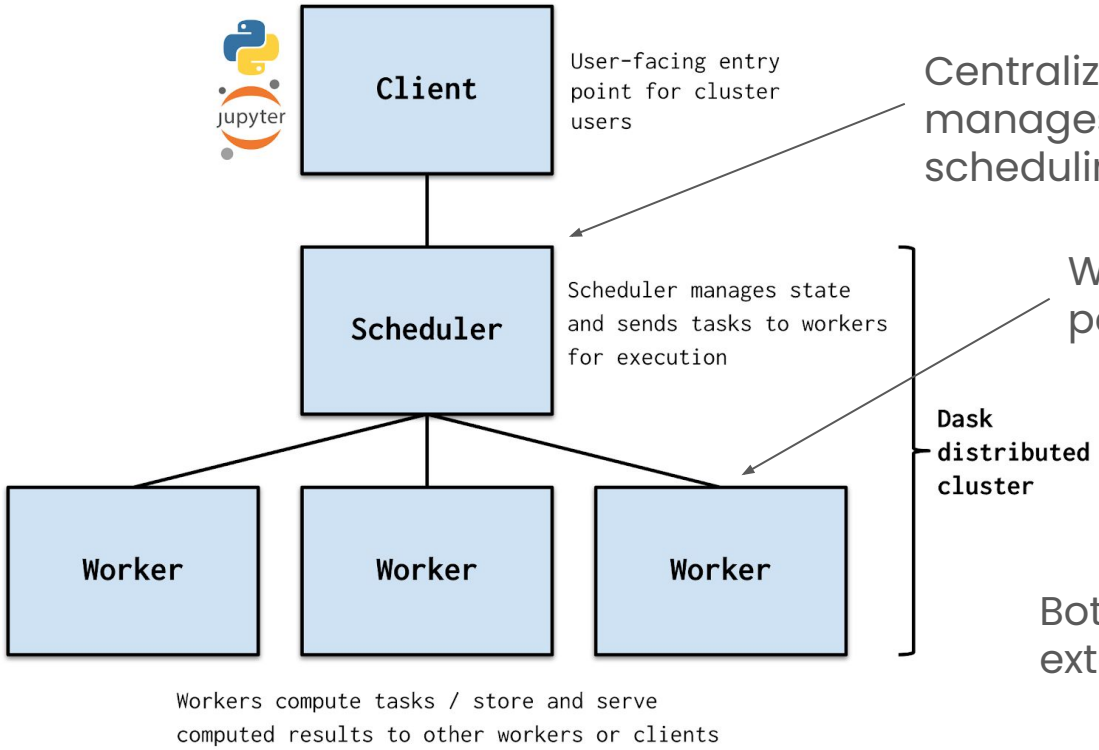
```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```



Dask task scheduling and execution

- Dask schedules tasks for execution onto one or more cores on one or more machines
- Dask provides several scheduler options:
  - A local synchronous scheduler, executing tasks serially
  - A local multithreaded scheduler, executing tasks on a threadpool
  - A local multiprocessing scheduler, executing tasks on a pool of processes
  - The distributed scheduler, executing tasks on a cluster of machines (can also be run locally)
- The distributed scheduler must be used when workloads exhaust the resources of a single machine

Distributed Dask Architecture



Centralized global scheduler that manages cluster state and implements scheduling policies

Workers share data over peer-to-peer TCP connections

Both the scheduler and workers are extendable via plugins

anyscale

The pitfalls of Dask Distributed

Descartes Labs

- Global scheduler is both a bottleneck and a single-point of failure
  - No scheduler fault-tolerance
  - Maintenance/introspection of cluster/task state adds task scheduling overhead
  - Scheduler is written in Python, and inherits Python's performance characteristics and resource footprint
  - Can't be natively scaled, would have to resort to sharding schedulers
- Clients are designed for end-users, not for usage within a larger production system
- Only nascent support for stateful execution

Why would Ray be a good fit for Dask workloads?

Ray Architecture Key Paradigms

Descartes Labs

Decentralized, peer-to-peer distributed scheduling	No global scheduler bottleneck
All cluster/task state centralized in a Global Control Store (backed by Redis), making decentralized schedulers stateless	Cluster/task state introspection shouldn't affect scheduling throughput
Distributed in-memory object store used for peer-to-peer data communication and zero-copy intra-node reads	Faster local data sharing between tasks
Support for fault-tolerant stateful actors	Great support for stateful execution
Resource-aware local-first scheduling policies with schedule decision caching	Smart distributed scheduling

Dask-on-Ray

A Dask scheduler that runs Dask tasks on Ray

anyscale

Dask-on-Ray API

anyscale

Dask-on-Ray API

This:

```
result = some_dask_collection.compute()
```

turns into this:

```
ray.init()  # connect to Ray cluster
dask.config.set(scheduler=ray_dask_get)
result = some_dask_collection.compute()
```

Pretty simple!

anyscale

Dask-on-Ray Example

```
# Start ray.
ray.init()

# Set the scheduler to ray_dask_get in your config so you don't have to specify
# it on each compute call.
dask.config.set(scheduler=ray_dask_get)


# Create a Dask DataFrame.
df = dd.from_pandas(pd.DataFrame(
    np.random.randint(0, 100, size=(1024, 2)),
    columns=["age", "grade"]), npartitions=2)

# The Dask-on-Ray scheduler submits the underlying task graph to Ray.
df.groupby(["age"]).mean().compute()
```

anyscale

Custom Shuffle Optimization

Custom optimization for Dask DataFrame shuffling, taking advantage of Ray's ability to have multiple return objects for a single task

```
with dask.config.set(
        scheduler=ray_dask_get,
        dataframe_optimize=dataframe_optimize):
    df = ...
    # Setting max_branch to infinity is required in order for our
    # optimization to work.
    df.set_index(
        ["age"], shuffle="tasks", max_branch=float("inf")).head(
            10, npartitions=-1)
```

anyscale

Ray-native Callbacks

Ray-native callback API for hooking into the Ray task submission and execution lifecycle

`ray_presubmit(task, key, deps)`:
Run before submitting a Ray task.

`ray_postsubmit(task, key, deps, object_ref)`:
Run after submitting a Ray task.

`ray_pretask(key, object_refs)`:
Run before executing a Dask task within a Ray task.

`ray_posttask(key, result, pre_state)`:
Run after executing a Dask task within a Ray task.

`ray_postsubmit_all(object_refs, dsk)`:
Run after all Ray tasks have been submitted.

`ray_finish(result)`:
Run after all Ray tasks have finished executing and the final result has been returned.

Persisting

Support for the persist() inlined-futures semantics, similar to Dask Distributed

```
# This submits all underlying Ray tasks to the cluster and returns a Dask array
# with the Ray futures inlined.
d_arr_p = d_arr.persist()

# Notice that the Ray ObjectRef is inlined. The dask.ones() task has been
# submitted to and is running on the Ray cluster.
print(dask.base.collections_to_dsk([d_arr_p]))
# {('ones-c345e6f8436ff9bcd68ddf25287d27f3',
#   0): ObjectRef(8b4e50dc1ddac855ffffffffffffffffffffffff0100000001000000)}

# Future computations on this persisted Dask Array will be fast since we already
# started computing d_arr_p in the background.
d_arr_p.sum().compute()
d_arr_p.min().compute()
d_arr_p.max().compute()
```

anyscale

Dask-on-Ray Implementation

anyscale

Dask-on-Ray Scheduler Implementation

Given that Ray tasks are just exported (pickled) Python functions…

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

and Dask task graphs are just function application graphs on (pickled) Python functions…

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

if we convert each Dask task's Python function to a Ray remote function…

```
ray_dsk = {key: rayify(task) for key, task in dsk.items()}

def rayify(task):
    if not istask(task):
        return task
    remote_func = ray.remote(task[0])
    return [lambda *args: remote_func.remote(*args)] + task[1:]
```

we can execute any Dask graph on Ray!

anyscale

User Stories

What are the use cases for Dask-on-Ray?

Large-scale Xarray Workload at Amazon

Process 3.3 PiB of multidimensional data, generating 15 TiB of output.

Experimented with both Dask Distributed and Dask-on-Ray.

Dask-on-Ray successfully scaled up to a **7x larger cluster** on AWS than Dask Distributed.

Dask-on-Ray achieved a **4x higher throughput per core** than Dask Distributed.

Overall, this resulted in a **13.5x faster processing** of their workload (14.4 days vs 195 days) and a **4x reduction in cost** (saving ~$700k).

Check out the blog post for more details!

anyscale

Takeaways

Ray can handle large cluster and data scales

Ray's ecosystem has several great data processing integrations, such as Dask-on-Ray

Resources

- Ray docs
- Dask-on-Ray docs
- Blog post on Amazon xarray workload

Thank You

anyscale