# Distributing Python Jobs using Ray and Dask

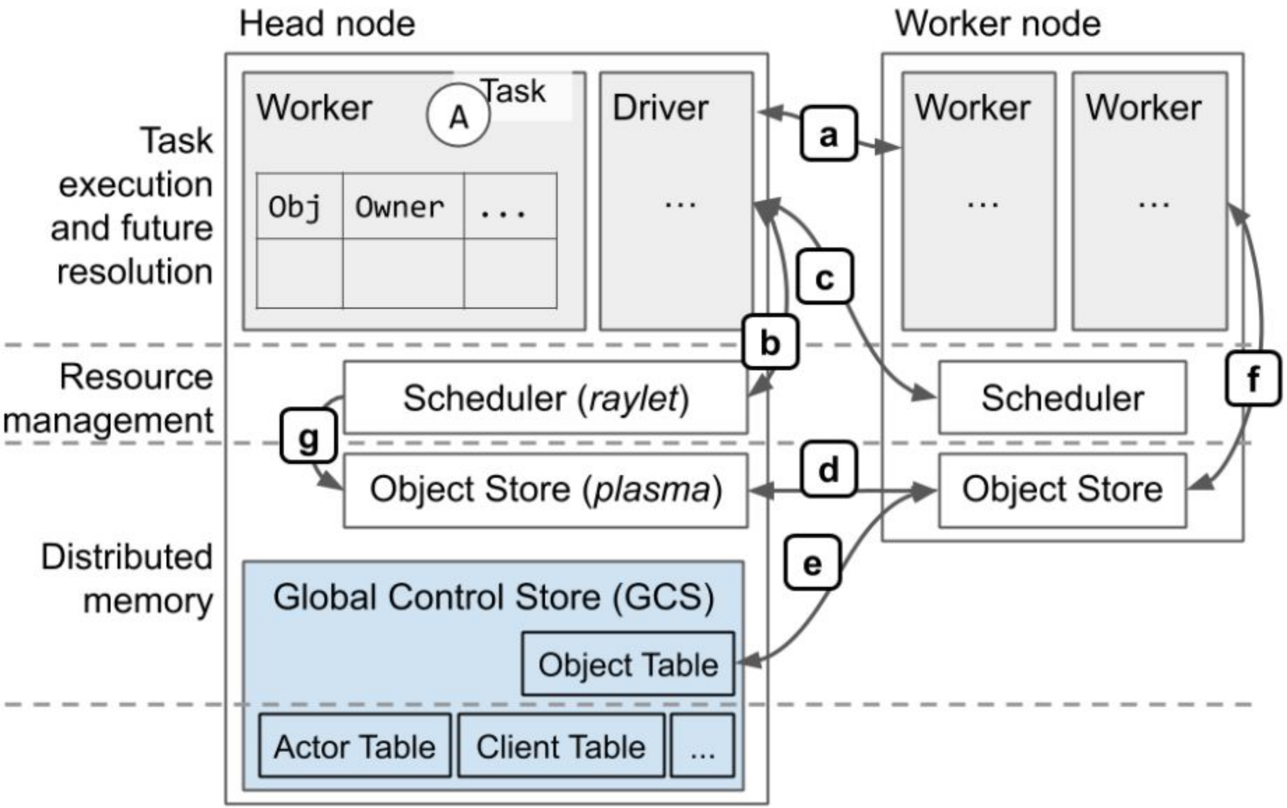
Vikash

# Ray

Ray provides a simple API for building distributed applications.

a. Providing simple primitives for building and running distributed applications.
b. Enabling end users to parallelize single machine code, with little changes.
c. Including a large ecosystem of applications, libraries, and tools on top of the core Ray to enable complex applications.

# Ray Architecture



# Ray Architecture

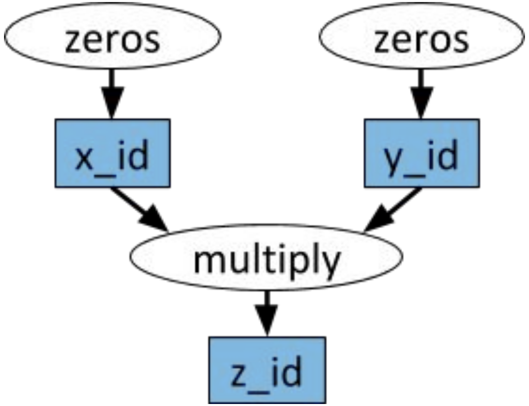
1. Task submission.
2. Local task scheduling.
3. Remote (distributed) task scheduling.
4. Distributed object transfer. Objects are stored in a node's object store.The object store is implemented with Plasma, part of PyArrow.
5. Metadata lookup in the global control store (GCS) for objects in the distributed object store and actors in workers, such as their locations. The object table holds the object metadata.
6. Storage and retrieval of objects created through ray.put and ray.get

# Ray Components

1. Ray cluster
2. Ray Serve: Scalable and Programmable Serving
3. RLlib: Scalable Reinforcement Learning
4. Tune: Scalable Hyperparameter Tuning
5. RaySGD: Distributed Training Wrappers
6. Modin (Pandas on Ray)
7. RayDP (Spark on Ray)
8. Distributed multiprocessing.Pool

# Dynamic Task Graphs

The underlying primitive in a Ray application or job is a dynamic task graph.



# Dynamic Task Graphs

```
@ray.remote
def multiply(x, y):
    return np.dot(x, y)
@ray.remote
def zeros(size):
    return np.zeros(size)
```

# Dynamic Task Graphs

```
# Start two tasks in parallel. These immediately return futures and the tasks are executed in the background.

x_id = zeros.remote((100, 100))

y_id = zeros.remote((100, 100))

# Start a third task. This will not be scheduled until the first two tasks have completed.

z_id = multiply.remote(x_id, y_id)

# Get the result. This will block until the third task completes.

z = ray.get(z_id)
```

# Python Functions to Ray Tasks

```
@ray.remote
def data_processing(input,...):
     #Processing ...
    return value

ref = data_processing.remote(....)
print(ray.get(ref))
```

# ray.get() vs. ray.wait()

- Calling ray.get(ids) blocks until all the tasks have completed.
- If some of the tasks, where some will finish more quickly than others then ray.wait() is recommended for such use cases.

```
def multple_tasks(input):
    refs = [task refs..]
    still_running = list(refs)
    while len(still_running) > 0:
        finished, still_running = ray.wait(still_running)
        finished_tasks = ray.get(finished)   # won't block
```

# Ray Actors

- It's a *message-passing* model, where autonomous blocks of code, the actors, receive messages from other actors asking them to perform work or return some results.
- Implementations provide thread safety while the messages are processed, one at a time.
- Many messages might arrive while one is being processed, they are stored in a queue and processed one at a time, the order of arrival.
- There are many other implementations of the actor model, including Erlang, the first system to create a production-grade implementation, initially used for telecom switches, and Akka, a JVM implementation inspired by Erlang.

# Python Class to Ray Actor

```
@ray.remote
class Counter:
    def __init__(self):
        self.label = 'Counter'
        self.count = 0
    def next(self):
        self.count += 1
        return self.count
```

# Create Actor Instance

- Construct actor instances with my_instance = Counter.remote(...).
- Call methods with my_instance.next.remote(...).
- Use ray.get() and ray.wait() to retrieve results, just like you do for task results.

**Detached Actors**

- they are designed to be long-lived actors that can be referenced by name and must be explicitly cleaned up.
- ray.kill(instance) to be removed

# Demo

- A very simple Distributed Feature Store implementation *“tinystore”* using ray.
- Functions
  - Register Features.
  - Register Aggregation functions.
  - Query Features.
  - Apply Aggregation functions.

# demo