DSC-204A: Scalable Data Systems, Winter 2024

Guest Lecture (03/04/2024): A Brief History of the Ray Ecosystem

Lecturer: Hao Zhang *Scribe: Yuancheng Cao, Yuwei Zhang, Yuxuan Zhang*

1 What is Ray?

1.1 Why Ray?

Two main trends are highlighted. First, the compute demands for AI are exploding, necessitating the need for **scale**. This is evident from the well-known graph with time on x-axis and the number of floating-point operations (FLOPS) required to train a model on the y-axis, which shows an exponential growth. Nowadays, not only are models becoming larger, but there is also more data that needs to be processed by these models. Second, the diversity of AI applications is also exploding, requiring **flexibility**. This means that machine learning models are being integrated into more legacy applications. Furthermore, multi-modality models are becoming increasingly popular today. These two trends are expected to continue in the future.

1.2 Ray: A Unified System for ML

On a single node, Python libraries are the key to application development due to two main advantages:

1. **Performance**: Libraries are often optimized with native code, ensuring good performance without requiring extensive optimization efforts from developers.
2. **Developer productivity**: Libraries can be easily composed together by simply making function calls, enabling developers to leverage existing functionality efficiently.

The motivation behind a unified system like Ray is to bring the ease of use and developer productivity of composing Python libraries to the distributed setting. For instance, on a laptop, developers would import various libraries for different tasks such as training, reinforcement learning, inference, and more. Putting these libraries together is straightforward, involving simple import statements and making function calls between them. This approach facilitates development as the libraries are already optimized for performance, allowing developers to focus on their application logic rather than low-level optimizations.

However, in the distributed setting, there is a need to address a range of domain-specific problems and challenging systems issues, such as distributed scheduling and fault tolerance. Solving these problems requires domain-specific solutions, as the requirements for fault tolerance in training, for example, are very different from those in other domains.

The idea behind distributed systems is that we now have a problem of scale. Having these distributed systems will be the trend in the future as well. However, the problem is that due to the difficulties in handling distributed systems challenges, we end up with a picture (as shown in Figure 1) where separate distributed systems are designed for each purpose. This is a very different scenario from the single node development story. When considering **developer productivity**, it is not just about stitching together

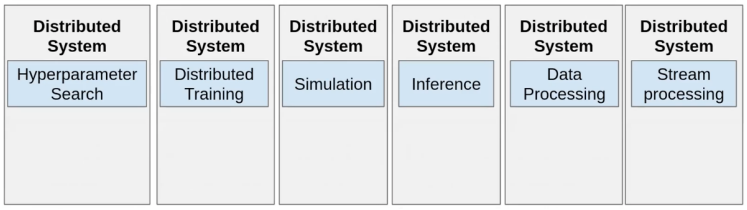

Figure 1: ML Ecosystem: Separate Distributed Systems

different libraries. Developers need to think about how to orchestrate execution across essentially different clusters and how to move data between them. This also brings **performance problems**. Developers need to consider how to move data efficiently between different systems, how to achieve end-to-end performance, and how to ensure that the system will be able to handle new domains in the future, making it future-proof.

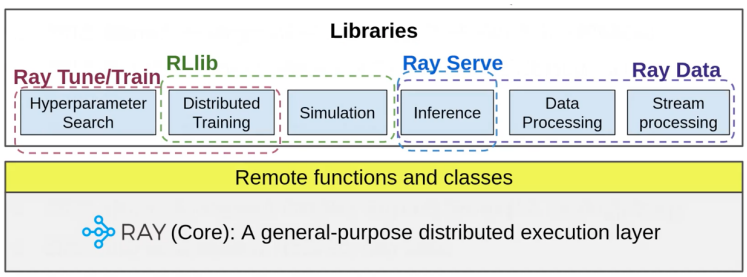

Figure 2: Ray Libraries

The motivation behind the development of the system was to build a general-purpose distributed execution engine that could handle different tasks in a unified system instead of having them siloed. This allowed turning these tasks into libraries again. The system now has libraries for various domains:

1. RLlib, introduced in 2017, is a library for reinforcement learning.

2. Ray Tune/Train, also introduced in 2017, is a library for hyperparameter search.

3. More recently, libraries for online inference and Ray data were added.

1.2.1 Audience Question: What is the advantage of unifying everything in Ray, rather than developing separate libraries?

Firstly, although the libraries could be considered separate, the main advantage is that there are no longer separate distributed systems. Operationally, this is crucial because it allows running everything on the same cluster, which is an important requirement for many industry users. Secondly, performance is enhanced by being able to move data across library boundaries without actually copying data between processes. This aspect of avoiding unnecessary data movement is really important. Additionally, another advantage relates to the effort required for systems development. As the application domain changes, a significant portion of the systems work can be reused for the new application when using a unified approach like Ray, reducing development efforts.

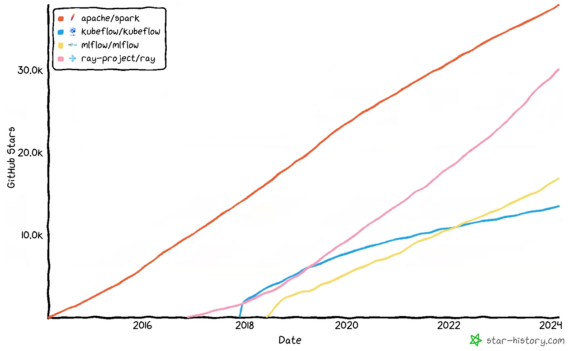

Figure 3: Ray Libraries

1.3 GitHub start History

Popular libraries like Apache Spark have garnered a large user base and have proven to be robust and reliable solutions. GitHub, while not a perfect measure of success or adoption, serves as a good enough proxy to gauge the popularity and health of these projects. When compared to other popular machine learning applications, Apache Spark and similar open source projects demonstrate their widespread use and active development communities.

1.4 History of Ray

The development of Ray has undergone a long journey, starting in 2016 when work began on version 0.1 at the RISELab at UC Berkeley. In 2017, the Tune (hyperparameter search) and RLlib (reinforcement learning) libraries were introduced. The Ray core was rewritten in C++ in 2018, and the first Ray paper was published at OSDI'18. In 2019, Anyscale was founded, and a second rewrite of the Ray core commenced. Ray V1.0 was released in 2020, marking the first major release, along with the introduction of the Serve (ML serving) library and the first Ray Summit. The Ray v1.0 paper appeared at NSDI'21 in 2021, and Ray Data was also introduced that year. Ray v2.0 was released in 2022, the same year OpenAI released ChatGPT. In 2023, Ray achieved the CloudSort world record, outperforming Spark.

1.5 The Ray API

The Ray API, also known as the Ray core, is the underlying distributed system that all Ray libraries are built upon. At its core, the API is quite simple and is based on the idea of allowing arbitrary Python functions and classes to be executed remotely, providing a general-purpose approach.

Firstly, there are functions, which are just normal Python functions. When programming with Ray, instead of receiving the actual value upon calling these functions, an object reference (a future) is returned. This future represents a value that will be evaluated asynchronously on a remote worker, and it can appear anywhere in the system.

Secondly, Ray introduces the concept of actors, which are similar to functions but stateful. Actors are essentially remote stateful instances, where a copy of the state (e.g., a counter) is stored on a worker, which could be located anywhere in the distributed system.

For example, o1 is a future, which represent the eventual value computed by `f`, and o1 also is remote

references, which represent values that may be stored in Ray's distributed object store on remote nodes (as shown in Figure 4).

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])
```

Figure 4: Illustration of **o1** as a future and a remote reference in Ray's distributed object store

1.6 2018: Ray pre-1.0 Architecture

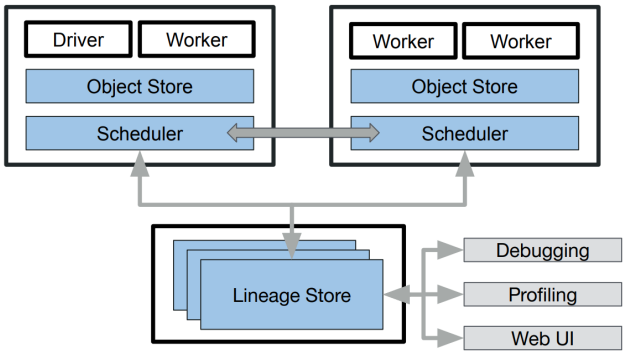

Figure 5: Ray pre-1.0 Architecture

For the first 2 or 3 years of Ray project, the architecture had several aspects that were important. One part was the object store, a distributed object store that stored the task inputs and return values. Every time a remote function was called and an object reference was obtained, the actual value for the object would have been stored in this object store. The idea was that, since this was a distributed object store, in order to get values from one place to another, they could just be copied. And since the values were immutable, there was no need for synchronization.

Another important aspect, which was discussed extensively in the OSDI paper, was the idea of a **global control store (GCS)**. This was a distributed key-value store that held all of the system state, including descriptions of the tasks to run. For example, if an object was lost from one of the object stores and needed to be recreated, if it was known that all the tasks were deterministic and idempotent, then the tasks could be re-executed. All the information about which tasks to re-execute would have been stored in the GCS.

The rationale behind the GCS was that it should simplify many things because debugging tools and other components could connect to the GCS without needing to touch any other components. However, the downside of storing everything in the GCS was that it meant executing multiple messages to complete one task and get its result. For instance, if a driver submitted a task that executed on a worker on another node, the request would have to traverse a path through the GCS for scheduling the task and then back again

to get the result. This turned out to be a significant performance problem because every execution in Ray involved multiple messages, even for actors where the worker location was known, and even for very small objects where it would have been faster to execute them with an RPC and get the value back directly.

Another major problem was **fault tolerance**. In the original paper, the goal was to provide very strong guarantees in case of failures, such that if any nodes went down, the information in the GCS could be used to recover. However, there were a few issues with this approach. One problem was that, due to the general-purpose API with features like actors with internal state and nested objects, it was difficult to guarantee transparent recovery of everything. Another issue was the decentralized design, where all nodes were meant to be homogeneous and important information was stored in the GCS. This made the system quite unstable because, in the case of a failure, it was complicated to decide which node would recover a particular object.

Furthermore, an important missing feature at that time was automatic memory management. If an object reference was in scope, it was necessary to ensure that the value would appear somewhere in the cluster. Conversely, once the object went out of scope, its value should eventually be cleaned up. In the initial versions of Ray, the only mechanism relied upon was Least Recently Used (LRU) eviction from the object store once it filled up. However, this made it difficult to distinguish between machine failures and out-of-memory scenarios. If an object was not present in the cluster, it could be either because memory was exhausted or because the node holding the object had failed, and the system could not determine whether to recover the value or not.

1.7 Designing Ray v1.0

Problems:

- Decentralized design added significant **overhead**, especially for actor tasks, as it required touching multiple components to perform a single operation.
- The system's **complexity** created instability under resource load and failures.
- Automatic memory management was needed for better stability, but adding it to the already complicated architecture would have introduced even more overhead and complexity.

→ These issues necessitated redesigning the metadata control plane for Ray 1.0.

Some parallel ideas:

- **Performance**: Reduce load on lower system components by having workers send tasks **directly** to each other via RPC, eliminating the need to go through multiple layers.
- **Reducing complexity**: Instead of a fully decentralized design where all system state was stored in the Global Control Store (GCS), keep the decentralized part but introduce a notion of metadata **ownership**.
- **Metadata ownership**: The owner should be the original reference holder (the worker that created the original ObjectRef). This was an obvious choice as it would enable automatic memory management. When a remote function is called, and an object reference is obtained, the caller becomes the owner of the metadata because they have all the necessary information about the object. However, when passing that reference to someone else, the ownership needs to be transferred.

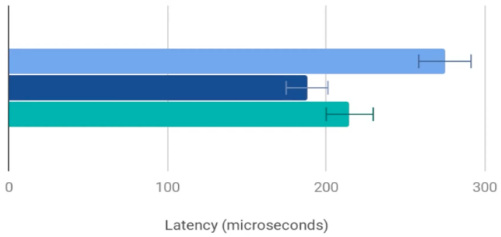

Figure 6: Actor Call Latency

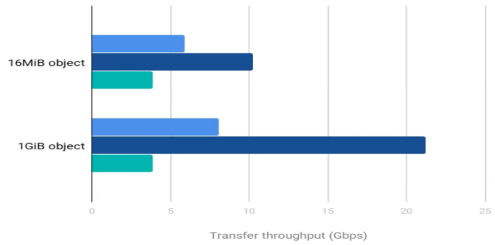

Figure 7: Object Transfer Throughput

1.8 2018: Key metrics leading up to Ray v1.0

In Figure 6, one crucial aspect was the performance of actor calls, specifically the latency. The goal was to achieve latency similar to gRPC 1.2 (Python) for small, direct worker-to-worker interactions. This was a significant improvement compared to the previous Ray 0.7 (OSDI architecture) and Ray 0.8 (first version after the initial rewrite).

In Figure 7, for larger objects that would be stored in the distributed object store, the objective was to achieve higher throughput than gRPC. The rationale behind this was that, by using Ray's distributed object store, it should be possible to share copies between workers, leading to better performance compared to gRPC, which requires copying objects whenever they are passed around.

Stability Changes: While harder to quantify, improvements in stability were also targeted through the introduction of features like task retries and automatic memory management.

1.9 2020: A distributed futures system for fine grained tasks

The system aims to achieve generality while imposing minimal overhead. By drawing an analogy with gRPC, which is capable of executing millions of tasks per second, the question arises whether a similar efficiency can be achieved for distributed futures. Distributed futures refer to futures whose values can be stored anywhere. The primary goal is to construct a distributed futures system that not only guarantees fault tolerance but also ensures low task overhead. The system facilitates applications that dynamically generate fine-grained tasks, offering a novel approach to task management and efficiency. Further details can be found in the referenced paper[1].

1.10 2020: Distributed futures introduce shared state

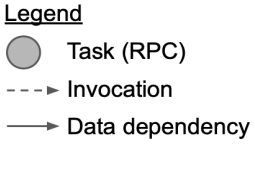

Figure 8: Legend of Graph

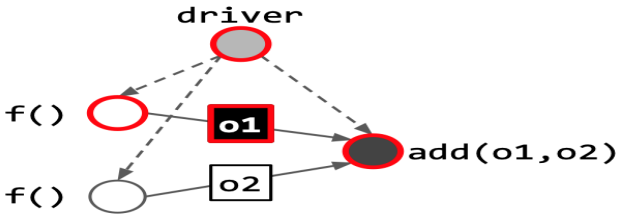

Figure 9: Shared state between different processes

Distributed futures introduce a challenging problem in distributed systems due to the shared state between different processes [refer to Figure 9]. Unlike RPC (Remote Procedure Call), where the returned value is

stored locally and has no connection to the remote process, distributed futures maintain a shared state among multiple processes. This shared state exists between the process that originally called the task, the process that created the value, the process that reads the value, and the physical location where the value is stored. Consequently, dereferencing a distributed future requires additional coordination and communication between the processes involved, adding complexity compared to the simpler RPC model.

1.11 Ownership

The existing solutions did not take advantage of the inherent structure of a distributed futures application, where task graphs are hierarchical and a distributed future is often passed within the scope of the caller. The insight was that by leveraging the structure of distributed futures applications, decentralization could be achieved without requiring expensive coordination.

The approach, called ownership, was based on the idea that the worker that calls a task owns the returned distributed future. For failure handling, each **worker** acts as a **"centralized master"** for the objects that it owns. This leverages the application structure, where the driver needs to be aware of tasks it creates, but not tasks created by other callers. Distributed futures usually have a specific scope, so objects created by the driver stay within that scope.

Ownership allows decentralizing failure handling, with each worker being a centralized master for a subset of objects instead of all objects. Supervision is used to handle owner failure, recursively cleaning up objects owned by a failed worker until the worker can be re-executed by its owner to recreate the objects. Regarding performance, ownership ensures **no additional writes** on the critical path of task execution, as task arguments and return values can be directly called from the caller to the worker, similar to a single RPC call. The system can also scale horizontally through **nested function calls**.

This ownership architecture was implemented in the first major release of Ray and is still in use, providing improvements not only in performance but also in stability by leveraging the inherent structure of distributed futures applications.

1.12 Today: When to use Ray Core?

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

Figure 10: Ray Features Comparison: Orchestration and Memory Management.

One way people use Ray Core is through coarse-grained orchestration, which means orchestration at the process level. In many cases, this could involve creating multiple actors that will run the job, but not necessarily orchestrating the application at a finer function level.

Another way people use Ray Core is for fine-grained orchestration. This means actually using direct function calls like `remote()` or `actor.remote()` in order to run the application. Additionally, there is the aspect of distributed memory management, which essentially involves anything that interacts with the distributed

object store shown earlier. Distributed memory management has been heavily used in data processing today, with the Ray Data library relying on this feature to store its datasets.

Finally, an important note is that because Ray Core is built as a unified system, there is a significant benefit when it comes to composing these different capabilities. One good example is combining Ray Data and Ray Train. Even though Ray Train only relies on Ray Core for coarse-grained orchestration, there is still an important benefit because Ray Data can be used alongside for data processing. This allows executing on a heterogeneous cluster where Ray Data runs on CPU cores and feeds data into a distributed training job, which is one of the key use cases highlighting the advantage of a unified system.

1.12.1 Audience Question: Further Clarification on the Distinctions Between Coarse-Grained and Fine-Grained

Fine-grained refers to splitting an application into very small units, such as functions that run in the range of 10 milliseconds. Coarse-grained, on the other hand, means having longer-lived processes, which could last for minutes or even hours. In the case of coarse-grained processes, there could potentially be additional orchestration happening within those processes. One example is provided in the context of Ray Train, which uses a pool of actors and performs placement group scheduling to assign GPUs to specific training tasks in a GPU cluster. However, a different framework like PyTorch Distributed is commonly used for the actual execution within the actors.

2 Ray Data: Scalable Datasets for ML

2.1 Ray Data is a flexible and scalable data processing library

Ray Data stands out in the data processing landscape due to its Python-native design that ensures ease of use and simple deployment via Ray Core. Key features include:

- **Ease of Use:** Python-native with easy deployment.
- **Transparent Scale:** Includes fault tolerance, resource management, and data partitioning.
- **Flexibility:** Pipelining across CPU and GPU tasks, and support for various data types.

Stephanie Wang emphasizes the library's automated handling of distributed data processing features, such as resource management and dataset partitioning. Additionally, the library's ability to handle CPU and GPU task pipelining offers significant use case flexibility, like data loading for machine learning.

Offline capabilities include large-scale shuffle operations, while online processing meets demands for low latency in tasks such as batch inference and vector database construction.

Wang's focus on data loading for training highlights the library's robustness in creating flexible data processing solutions.

2.2 Data loading for ML training

- Data loading needs to be fast, to maximize GPU utilization in Figure 11

Figure 11: Maximize GPU to Accelerate Data Loading

Figure 12: Scale to Large Datasets and Clusters

- Data loading needs to scale to large datasets and clusters. Large dataset must stream through memory and cluster must send data over the network in Figure 12.
- Data loading needs to be flexible, to support arbitrary preprocessing. Data can have different: storage, modality, preprocessing, memory footprint, ordering, ... in Figure 13

2.3 Ray Data is...

In summary, ray data is fast, scalable, and flexible. Streaming execution and shared-memory data loading could maximize the GPU utilization. The heterogeneous clusters and automatic failture recovery could increase the scalability of ray data. Finally, the query planner for building arbitrary data preprocessing pipelines could ray data to be compatible and flexible among multiple different input and applications.

2.4 Ray Data design

In Figure 14,

- Ray cores, as computing units, implement workers.

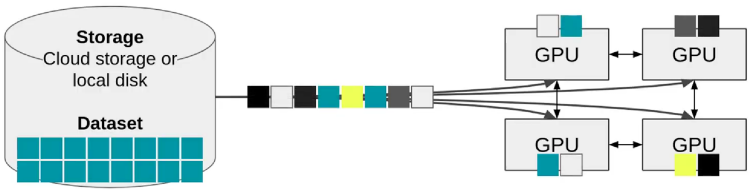

Figure 13: Flexibility of Ray Data

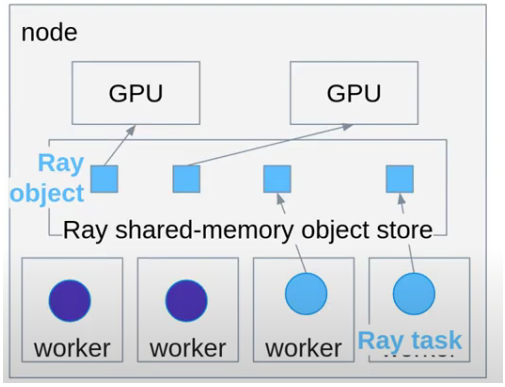

Figure 14: Ray Data Design

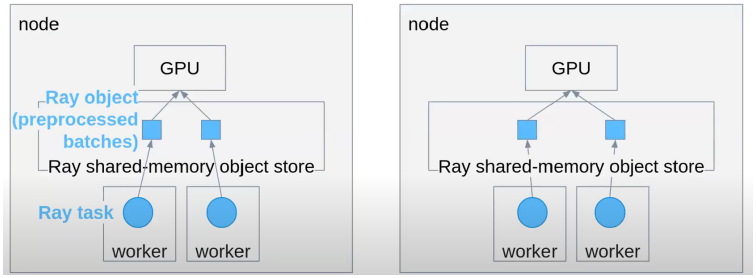

Figure 15: Ray Data with Distributed Trainers

- wockers put data in shared memory
- work is broken into smaller "tasks" scheduled by ray and task outputs are spread on the fly among GPUs.

Additional Info of Ray Design:

data load and processing: Compared to multiprocessing, ray copy preprocessed data in shared memory and ray core task overhead.

data return: ray will automatically partition data; the scheduuler can control execution to dynamically load-balance and limit memory usage; recover from failures without having to restart

2.5 Ray Data distributed trainers

Ray data routes batches based on data locality and load-balancing in Figure 15

2.6 Caching Ray Datasets with `ds.materialize()`

Data can be cached at any stage of preprocessing and ray core automatically spills to disk to avoid out of memory.

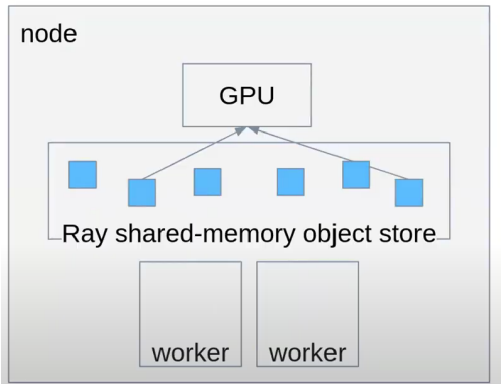

Figure 16: Caching Ray Datasets

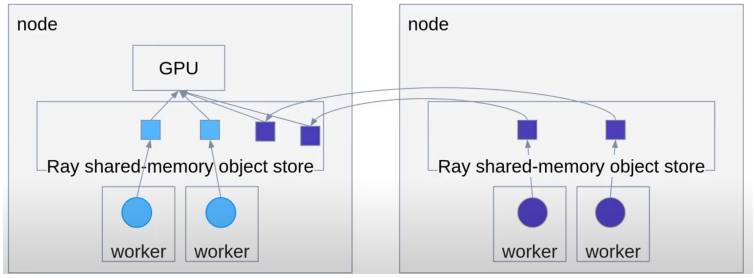

Figure 17: Ray Data with Heterogeneous clusters

2.7 Ray Data with heterogeneous clusters

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

2.8 Data loading for ML training features

The computing system described provides a robust infrastructure for both single-node and distributed operations. Key features include:

- **Single-node and Distributed Operations:**
  - Automatic partitioning of datasets and load-balancing across workers for efficiency.
  - Enforced memory limits to prevent overconsumption of resources.
  - Capability to recover from failures without the need to restart the entire training process.
  - In-memory and on-disk caching of materialized datasets for quick access.
- **Distributed Features:**
  - **Heterogeneous Clusters:** Enables the separation of CPU-based data preprocessing from GPU-based training tasks.
  - **Locality-Based Scheduling:** Optimizes task assignments based on data location.

– **Autoscaling Clusters (Soon):** Upcoming feature for dynamic scaling of resources based on workload demands.

The emphasis on *Flexibility* suggests that the system is designed to adapt to various operational demands, providing a versatile environment for computing tasks.

3 Conclusion

In this document, we have outlined the multifaceted features of Ray Data, a state-of-the-art data processing library that brings together ease of use, transparent scaling, and unmatched flexibility. Ray Data excels with its Python-native interface, making deployment straightforward and seamless on Ray Core. Its capabilities extend to accommodate fault tolerance, advanced resource management, and sophisticated data partitioning strategies.

The library is engineered for speed to maximize GPU utilization, scales adeptly to handle large datasets and distributed clusters, and provides the flexibility necessary to support arbitrary preprocessing needs. This is further enhanced with offline and online processing capabilities to meet a wide array of ML training demands.

Ray Data's design philosophy underpins its distributed training features, integrating shared-memory data loading, streaming execution, and the ability to recover from failures without disrupting ongoing processes. The heterogeneity of cluster support and the implementation of a query planner underscore its flexibility and compatibility across numerous applications and input types.

Finally, Ray Data's innovative caching strategy and the intelligent scheduling of tasks across heterogeneous clusters ensure that data is where it needs to be, when it needs to be there, thus optimizing the overall efficiency of ML training.

In conclusion, Ray Data is not just a tool but a comprehensive solution that addresses the critical aspects of machine learning workflows: speed, scale, and flexibility. Its carefully designed features enable practitioners to streamline the data-to-model pipeline, reduce computational overheads, and focus on achieving the highest levels of model performance. With Ray Data, the future of scalable and flexible machine learning is not just envisioned; it is realized.

3.1 Audience Question 1: Understanding Ray Data and Shared Memory Overhead

The questioner inquires about the possibility of reducing Ray Data overhead by utilizing a shared memory layer, specifically comparing the shared memory implementations of multiprocessing and Ray Data. The lecturer clarifies that Ray does indeed use shared memory, particularly native shared memory for its operations, contrary to some misconceptions that it relies on Redis for object storage. Redis is only used for metadata within Ray. The shared memory object store in Ray is managed directly, which results in similar overheads to multiprocessing when considering the shared memory aspect alone. The main difference lies in the optimization potential through better serialization formats, with Apache Arrow highlighted for enabling zero-copy deserialization for numerical data types.

3.2 Audience Question 2: Data Orchestration and ML Model Training

The discussion moves to the comparison between multiprocessing data loaders and Ray's approach, particularly in the context of machine learning model training. The questioner is curious about the orchestration

overhead and its implications for training. The lecturer highlights that Ray already incorporates specialized designs for LLMs, utilizing both process-level and function-level orchestration. However, challenges remain, especially around distributed memory management and execution efficiency for tasks closely aligned with LLM operations. Significant system-level adjustments are necessary to cater to the specific demands of LLMs, including managing GPU memory explicitly due to the limitations of shared memory object stores which are tailored more towards host memory.

3.3 Audience Question 3: Specialized Design for Large Language Models (LLMs)

The discussion shifts towards whether there should be specialized designs for LLMs, considering their unique requirements for training and inference. The lecturer highlights that Ray already incorporates specialized designs for LLMs, utilizing both process-level and function-level orchestration. However, challenges remain, especially around distributed memory management and execution efficiency for tasks closely aligned with LLM operations. Significant system-level adjustments are necessary to cater to the specific demands of LLMs, including managing GPU memory explicitly due to the limitations of shared memory object stores which are tailored more towards host memory.

3.4 Audience Question 4: Performance vs. Flexibility in Distributed Systems

A question arises regarding whether Ray sacrifices too much performance for the sake of flexibility, particularly in contexts demanding low latency and high efficiency. The lecturer acknowledges a trade-off between performance and flexibility, noting that while Ray excels in scenarios with tasks lasting tens of milliseconds or longer, it might not be the best fit for latency-sensitive operations or those requiring intensive accelerator orchestration. For such specialized needs, a different system design that prioritizes performance over dynamic scheduling might be more appropriate.

3.5 Audience Question 5: Use of Ray by OpenAI

The final question touches on how OpenAI utilizes Ray, seeking insights into the specifics of their implementation. While not having official information, the lecturer speculates that OpenAI likely leverages Ray for both process-level orchestration for managing GPU clusters and possibly for function-level tasks, though the extent of the latter's use remains uncertain. The primary application is believed to be in resource management and job scheduling within large-scale GPU environments.