Ray Workshop: Beginner to Intermediate

Nikolay Manchev

May 31, 2023

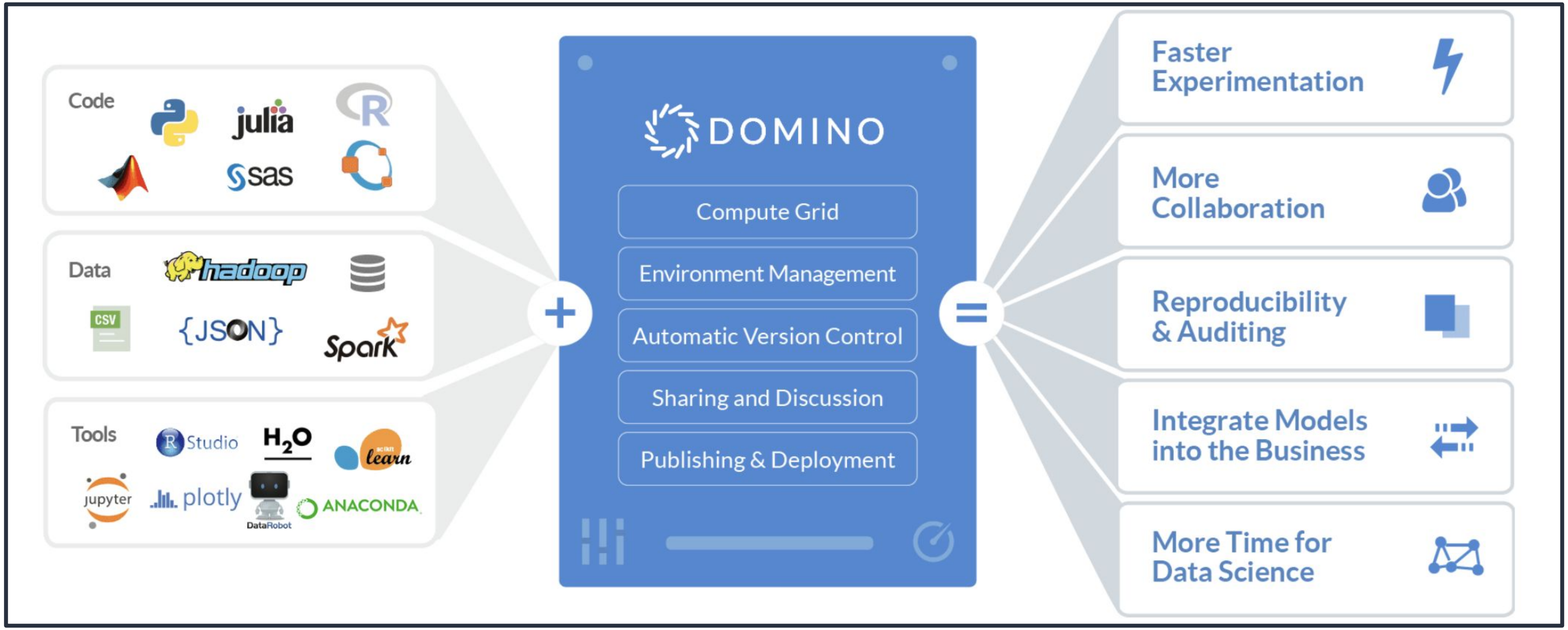
AGENDA

- Introduction
- Setting up Ray in Domino
- Connecting to Ray in Domino
- Submitting Remote Tasks
- Interpreting Logs and Errors
- Effectively Distributing Work
- Bonus: Autoscaling

Introduction

What is Domino?

Domino makes it easy to access scalable compute, update and share environments, keep track of your experiments and file changes, and easily deploy models



Field Data Science Practice

- Assist at all phases of the project:
    - Ideation, research and development, execution
- Expert data scientists and field engineers help you leverage Domino for success
- We provide consultation services:
    - Best practices for data science on Domino
    - Developing Proof of Concepts and Minimum Viable Products
    - Execution of complex projects to solve business problems
    - Knowledge transfer upon project completion

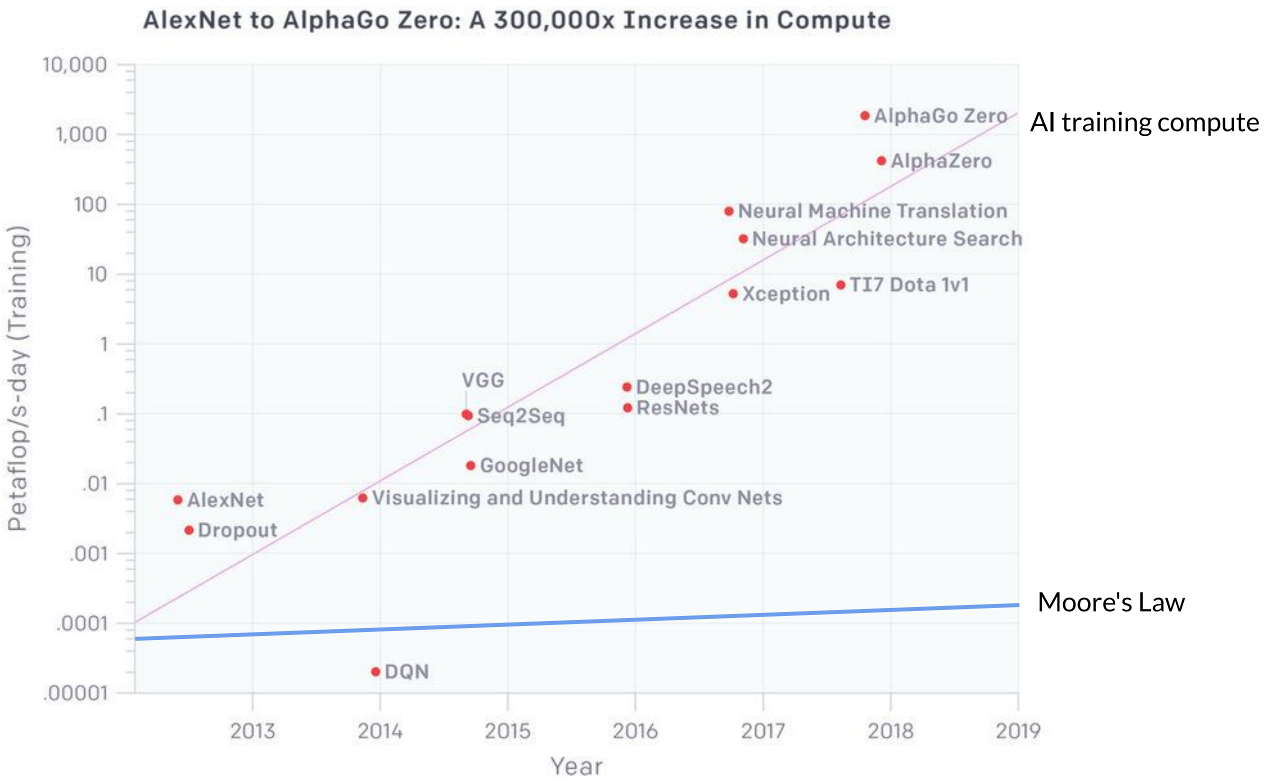
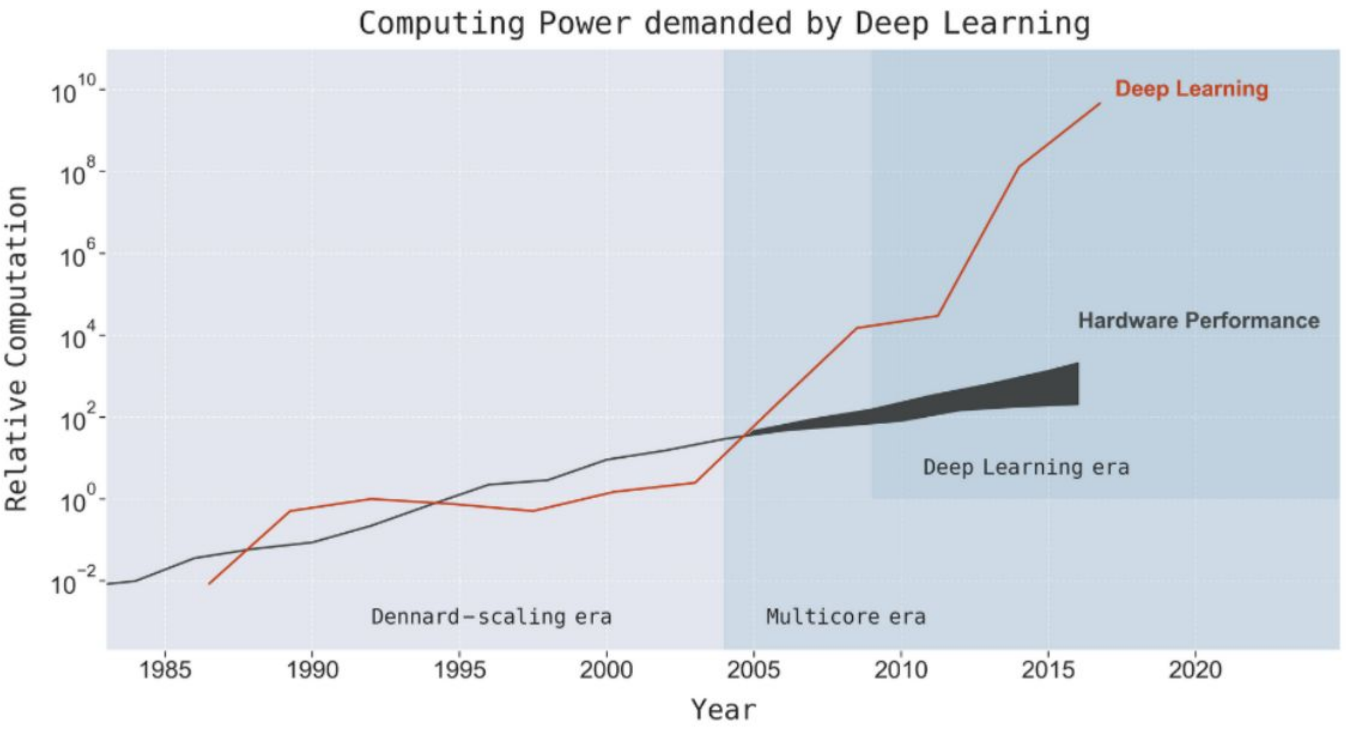


Illustration of the increasing compute demand from AlexNet in 2013 to AlphaGo Zero today; the exponential fit of the data points gave a doubling time 3.43 months, as given in Kozma, Robert & Noack, Raymond & Siegelmann, Hava. (2019). Models of Situated Intelligence Inspired by the Energy Management of Brains. 567-572. 10.1109/SMC.2019.8914064.



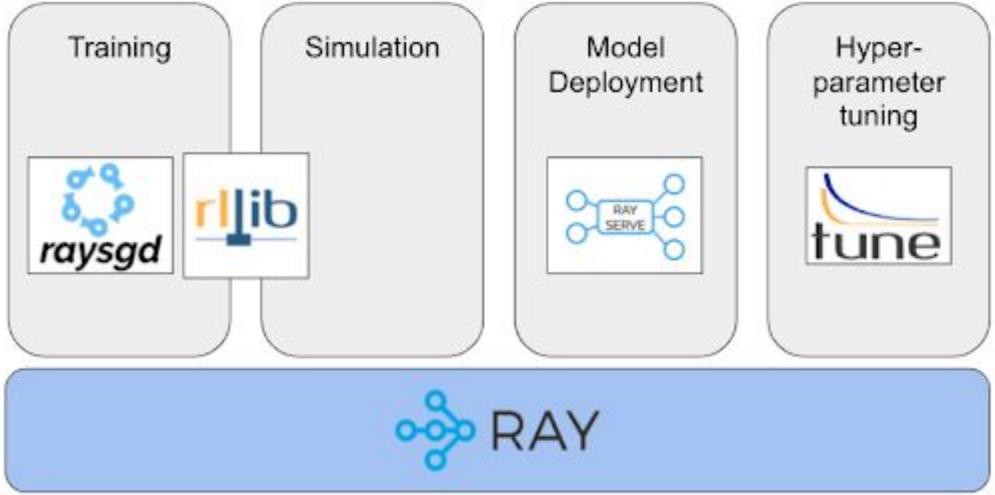
Deep learning models of all types (as compared with the growth in hardware performance from improving processors - Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: **Recording microprocessor history**. Queue, 10(4):10:10–10:27, 2012.), as analyzed by a) John L. Hennessy and David A. Patterson. **Computer Architecture: A Quantitative Approach**. Morgan Kaufmann, San Francisco, CA, sixth edition, 2019 and b)] Charles E. Leiserson, Neil C. Thompson, Joel Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. **There's plenty of room at the top: What will drive growth in computer performance after Moore's law ends?** Science, 2020.

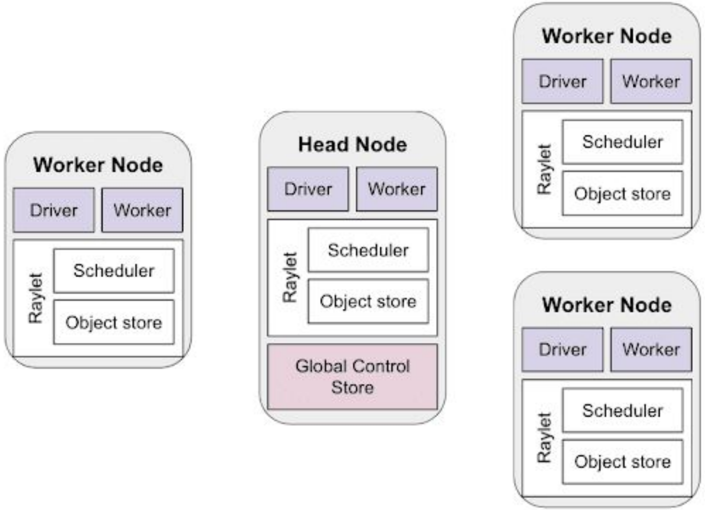
Figure from Neil C. Thompson1, Kristjan Greenewald2, Keeheon Lee3, Gabriel F. Manso, **The Computational Limits of Deep Learning**, arXiv:2007.05558v1 [cs.LG] 10 Jul 2020

DOMINO


Multithreaded programming
Theory
Actual

About Ray





- Simple, concise, and intuitive API
- Easy for people without distributed computing experience
- Flexible for a wide class of problems

Setting up Ray in Domino

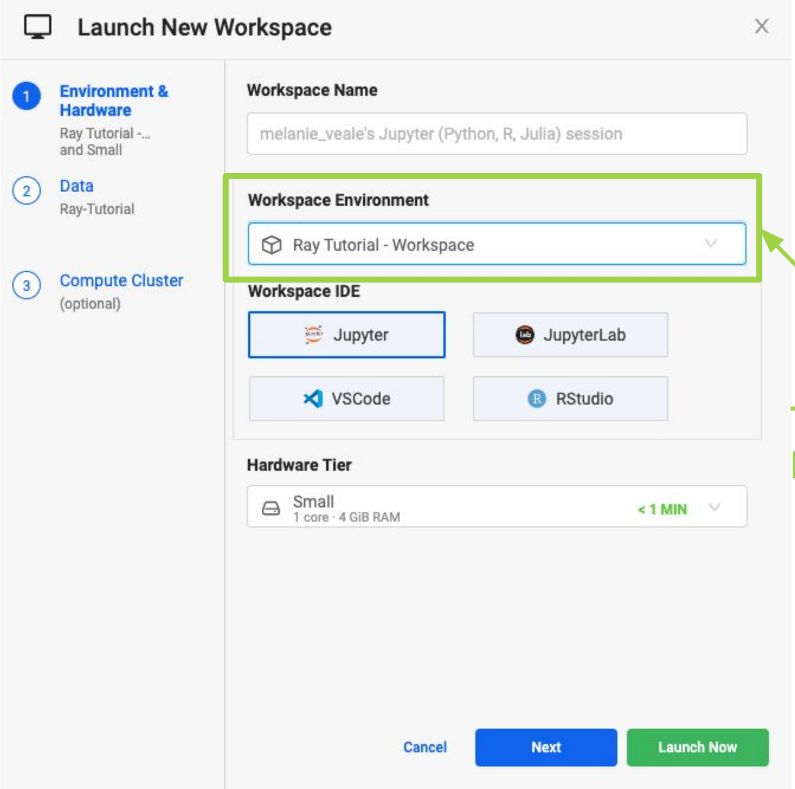
Ray Cluster setup in Domino

- Available for **Workspaces** and **Jobs**
- Remember there are **two compute environments** required
  - One for the workspace (a "normal" environment)
  - One for the cluster (must be labeled for use with Ray)
  - It is important for package versions to match between them!
- There are **three hardware tiers** to choose
  - One for the workspace
  - One for the cluster head node
  - One for the cluster workers
  - With Ray, the head node also functions as a worker, so unlike other cluster types you typically do not want to make the head node use smaller hardware
- (Don't worry about Autoscaling or Dedicated Local Storage for now)

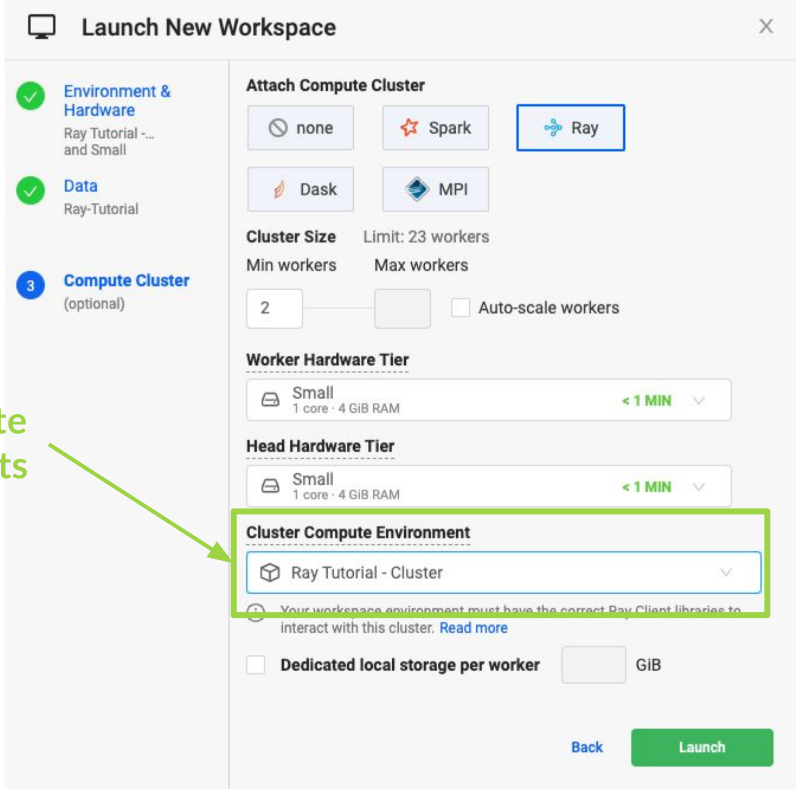
Activity

- Sign up for a Domino account: **tinyurl.com/ray-rev4**
- Fork the Tutorial project: **tinyurl.com/ray-project**
- Navigate to **Workspaces**
    - Create New Workspace
    - Choose **Ray Tutorial Workspace** compute environment, Small hardware
    - Skip Data
    - On compute clusters, choose Ray with 1 worker, no autoscaling
    - Choose Small hardware tier for both Worker and Head nodes
    - Choose **Ray Tutorial Cluster** compute environment
    - No Dedicated Local Storage
    - **Launch**
- When the workspace is ready, view the Ray Web UI

Ray Cluster setup in Domino

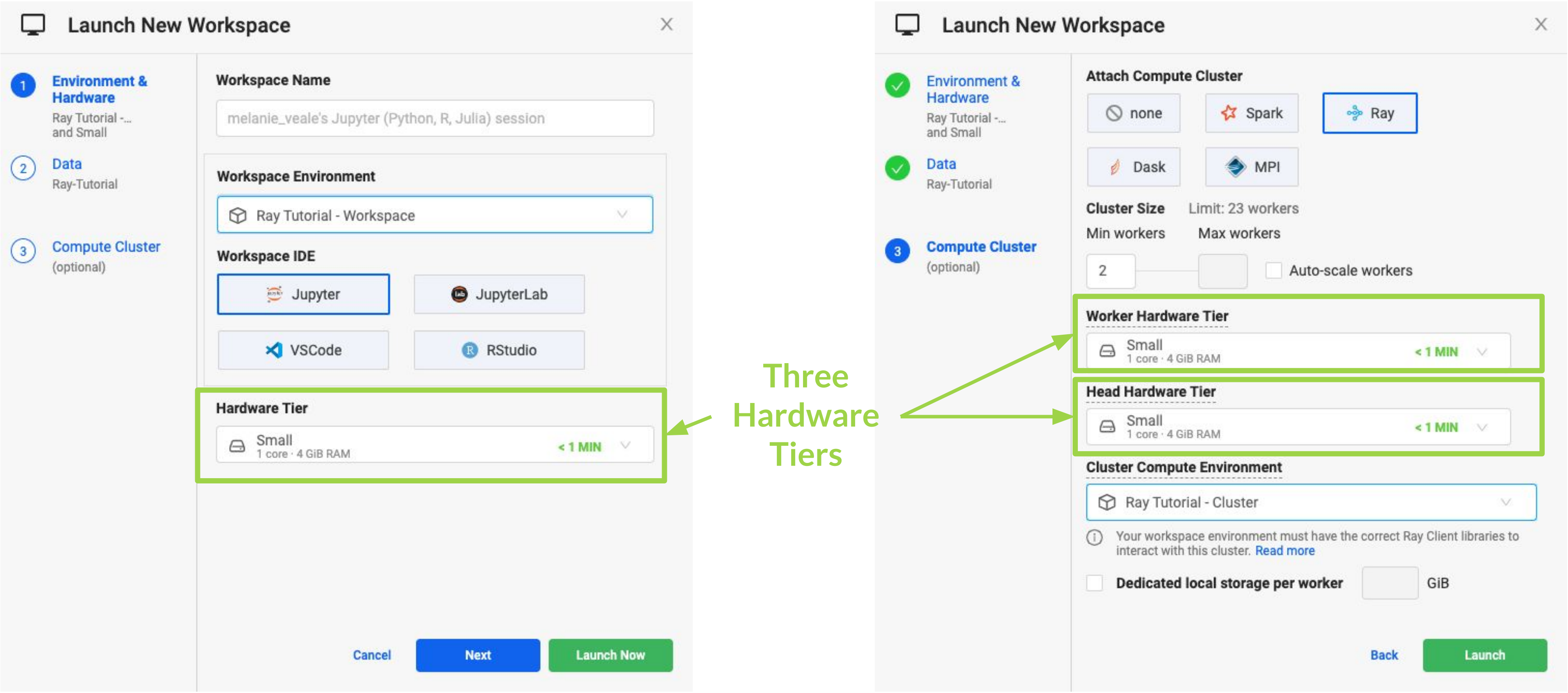


Two Compute Environments



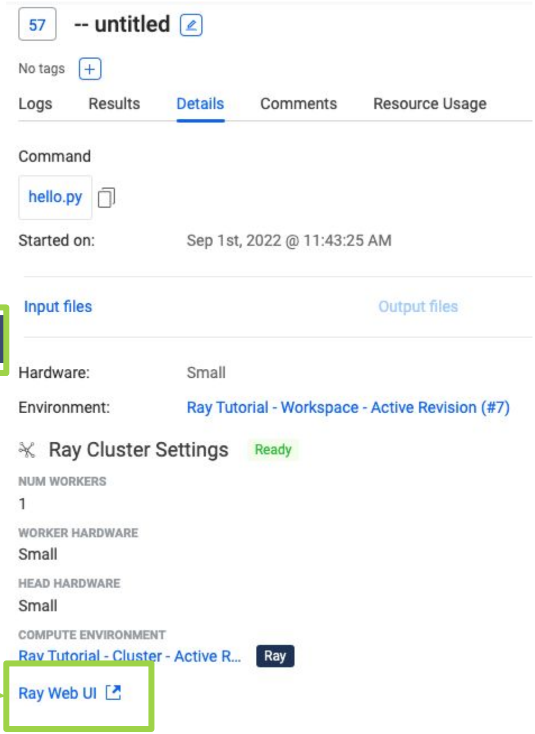
DOMINO

Ray Cluster setup in Domino



Ray Web UI in Domino

- Available for **Workspaces** and **Jobs**
- It is the standard Ray Dashboard
- Becomes available when **Head** node is ready
- Good place to check status of **Worker** nodes



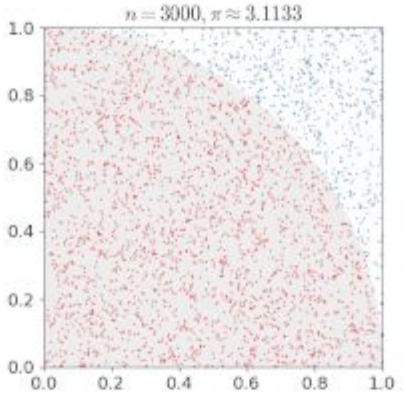
Available as a tab within a workspace

Link available from Details of a job

Our sample problem

We will be estimating the value of Pi using a Monte Carlo simulation. There is a good description on the [Wikipedia page] for Monte Carlo methods.

- Randomly scatter points onto a 1x1 square
- Calculate for each point whether it is within a unit circle
- The ratio of the number of points inside to the total is an approximation of the ratio of the area:
  pi ~ 4 * n_inside / n_total
- It is easy to split into sub-simulations that are independent of each other, which is critical for distributing the work across a cluster.



Connecting to Ray in Domino

Activity

- Open the beginner tutorial notebook and run the first few cells
- Compare the output to the contents of the Ray Web UI
- You may find it useful to duplicate the Workspace tab, then view your notebook and the Ray Web UI side-by-side on your screen
- Stop when you get to the section for “Submitting work to the Ray Cluster”

Connecting to Ray in Domino

The "wrong way" to connect to Ray:

```
ray.init()
```

The "right way" to connect to Ray:

```
service_host = os.environ["RAY_HEAD_SERVICE_HOST"]
service_port = os.environ["RAY_HEAD_SERVICE_PORT"]
ray.init(f"ray://{service_host}:{service_port}")
```

The "wrong way" will create a new "mini" Ray cluster local to your workspace machine. It can be surprisingly easy to miss this happening, because you can still run things in **Parallel**, so you may see things "working". But, you will not be **Distributing** work to the Domino Cluster.

Deliberately creating a local "mini" cluster can sometimes be useful for debugging purposes, especially suspected issues with package mismatch in the cluster.

Submitting Remote Tasks

Activity

- In the beginner tutorial notebook, continue running cells in the “Submitting work to the Ray cluster” section
- Pay particular attention to the handling of **futures**.
- Stop when you get to the “Logs, errors, and PIDs” section

Using the ray.remote decorator

If non-ray code looks like this:

```
def my_function(x):
    # do something
    return y

x1 = 10
y1 = my_function(x1)
```

Ray code will look like this:

```
@ray.remote
def my_ray_function(x):
    # do something
    return y

x1 = 10
y1_future = my_ray_function.remote(x1)
y1 = ray.get(y1_future)
```

Ray tasks have **immediate execution**; work starts as soon as submitted

Remember that getting results is a **blocking** operation

Interpreting logs and errors

Activity

- In the beginner tutorial notebook, continue running cells in the “Logs, errors, and PIDs” section
- Correlate what you see with the Ray Web UI
- Stop when you get to the “Parallelize and distribute work” section

Tips for interpreting logs and errors

- Printed messages from inside remote functions:
  - Are prefaced by task name and PID
  - Print as soon as they occur, wherever your current cell is
- Errors from inside remote functions:
  - May not print until you attempt to get the result
  - Compress the “relevant” stack trace into the final error message
- Errors resulting from running out of memory, or other Ray-specific problems, will present differently

Effectively distributing work

Activity

- In the beginner tutorial notebook, continue running cells in the “Parallelize and distribute work” section
- Read the code for each section in some detail - this is where things get exciting, and understanding the structure is important
- Stop when you get to the “Putting it all together” section

Tips for effectively distributing computations

- Always remember that ray.get() is a **blocking** call.
    - Do not try to get results within the same loop as tasks are submitted!
- Calling a remote function results in **immediate** execution.
    - This is true for remote Tasks, but may not be true everywhere in Ray (i.e. other Ray libraries for handling data may use **lazy** execution).
- Each remote function call incurs some **overhead**
    - Process items in **batches** to reduce total overhead costs.

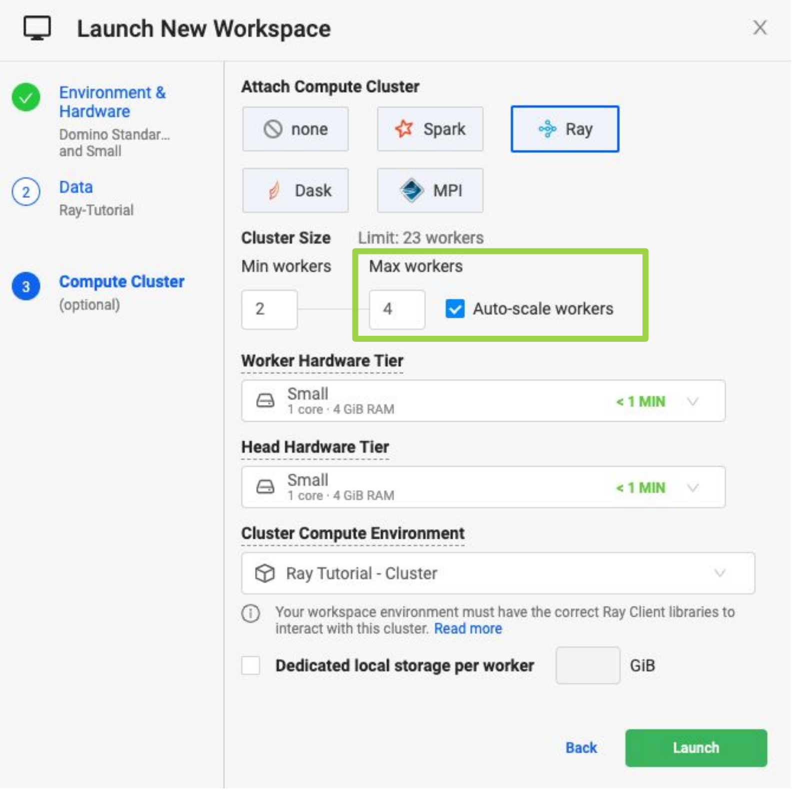
Bonus: autoscaling

Activity

- In the beginner tutorial notebook, follow the instructions for restarting the Workspace with Auto-scaling enabled
- Reopen the Beginner notebook, and start running cells from the “Putting it all together” section.
- Duplicate the tab again to get the new Ray Web UI side-by-side with the notebook, and watch the Auto-scaling add nodes
- Leave the notebook idle for 5 minutes or longer to see the Ray cluster scale back down

Auto-scaling Ray workers in Domino

- Clusters always start with the **minimum** number of workers
- If auto-scaling is enabled, they may add workers up to the **maximum**
    - By default, the cluster will scale up when **CPU** usage reaches **80%**
    - By default, the cluster will scale down when the usage drops below the threshold for **5 minutes**.
- Domino admins control the auto-scaling [behavior](), and whether auto-scaling is [available]() at all.



AGENDA

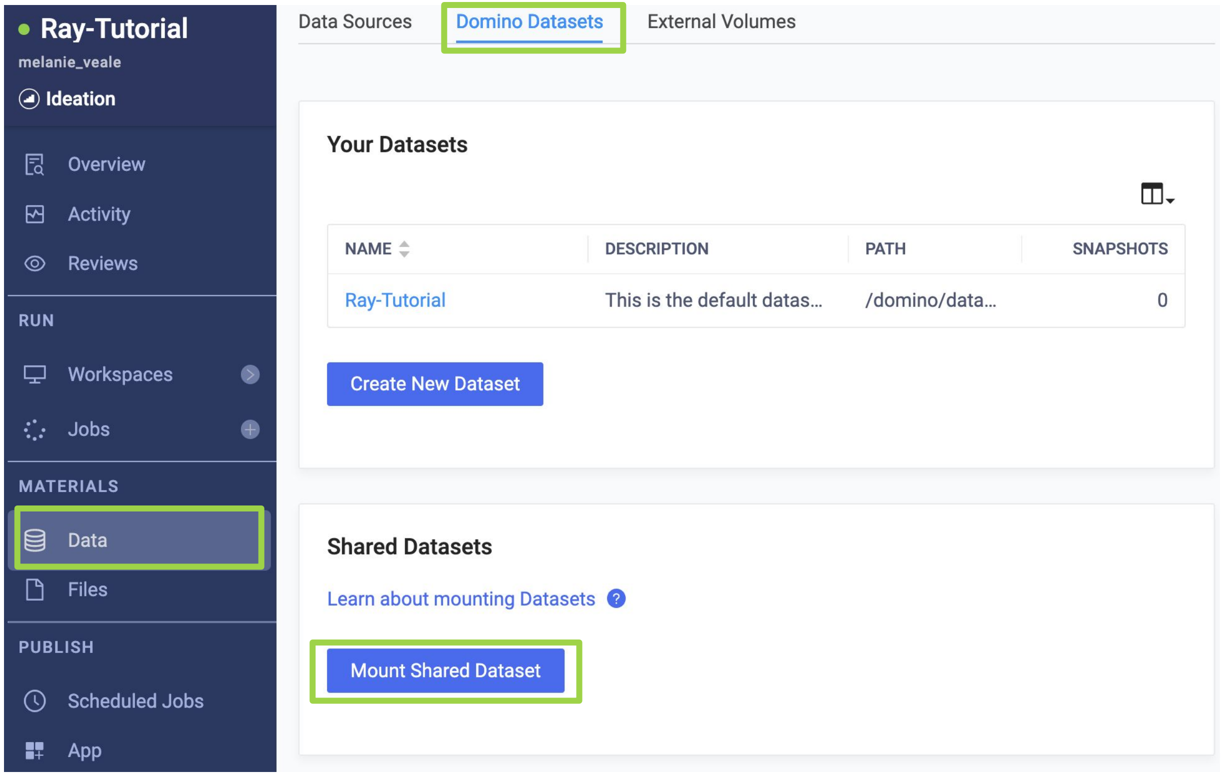
- Beginner Session Recap
- More Task Management
- Actors and Scheduling
- Object Store
- Ray Data
- Modin
- Tips for Large Data and Memory
- Bonus: Local Storage

Mounting Datasets

Datasets from other projects will be **read-only**

You must **restart** a running Workspace before a newly mounted Dataset will be accessible

We will use this in later sections of this tutorial



More Task Management

More options for Task management

- Use ray.wait before ray.get to process results as tasks finish
  - It is blocking until the number of requested tasks are finished
  - However, it does not replace ray.get - it still returns futures for those tasks
  - It always returns two lists, even if there are 0 or 1 tasks.

```
list_of_tasks = [my_function.remote(x) for x in list_of_inputs]
list_of_finished_tasks, list_of_unfinished_tasks = ray.wait(list_of_tasks)
list_of_results = ray.get(list_of_finished_tasks)
```

- Use ray.cancel to cancel tasks
  - If the task is finished, or has not yet started, this will happen “cleanly”
  - If the task is in progress, it may generate errors - these are benign IF your task is written to have no bad side-effects when interrupted.

Actors and Scheduling

Activity

- Run the next section of cells for “Ray Actors”
- Optionally, experiment with Autoscaling and more precision in the simulation
- Stop when you get to the section for “Object store and ray.put”

Remote classes in Ray (Actors)

If non-ray code looks like this:

```
class MyWorker:
    def __init__(self, x):
        # store state of x

    def do_work(self):
        # do something
        return y

w = MyWorker(x)
y = w.do_work()
```

Ray code will look like this:

```
@ray.remote
class MyWorker:
    def __init__(self, x):
        # store state of x

    def do_work(self):
        # do something
        return y

w = MyWorker.remote(x)
y_future = w.do_work.remote()
y = ray.get(y_future)
```

Remote classes in Ray (Actors)

To quote the Ray docs:

"[...] tasks are scheduled more flexibly, [and] if you don't need the stateful part of an actor, you're mostly better off using tasks."

- Actors are dedicated workers that "live" on a particular node.
- Actors will only execute one task at a time.

You are much more likely to interact with Actors indirectly, via other libraries.

Scheduling

Both Tasks and Actors can be scheduled, but the long-lived nature of Actors makes their scheduling less flexible and more likely to need intervention.

If all instances of an actor have the same resource requirements, it can be specified in the decorator; otherwise, resources can be specified at instantiation.

```
@ray.remote
class MyWorker:
...
w =
MyWorker.options(num_cpus=2)remote(x)
```

OR

```
@ray.remote(num_cpus=2)
class MyWorker:
...
w = MyWorker.remote(x)
```

This is only one way to do scheduling - see the [Ray docs] for many more options!

Object Store

DOMINO

Activity

- Continue running cells in the “Object store and ray.put” section
- Pay particular attention to the **Plasma** column in the Ray Web UI
- Understand which variables in the code represent **futures** or **object refs** versus the actual object data.
- Stop when you get to the “Loading large data with Ray Data” section

Using ray.put when passing large data to Tasks

Whenever large data is passed to a remote Task, it has to be copied to the cluster.
It is much better to copy the data once explicitly, then pass an **object ref** instead.
Ray automatically dereferences top-level arguments inside remote functions, so both of the following work with no change to the Task function code.

Not great:

```
# x is something large
y1_future = do_something.remote(x)
y2_future = do_something_else.remote(x)
```

Much better:

```
# x is something large
x_ref = ray.put(x)
y1_future = do_something.remote(x_ref)
y2_future =
do_something_else.remote(x_ref)
```

DOMINO

Delaying ray.get until final results

A similar principle applies when dealing with intermediate results.
If the results will be passed onto another Ray Task, there is usually no need to use ray.get on the intermediate variable at all.

Not great:

```
# y is something large
y_future = do_something.remote(x)
y = ray.get(y_future)
z_future = do_something_else.remote(y)
z = ray.get(z_future)
```

Much better:

```
# y is something large
y_future = do_something.remote(x)
z_future = do_something_else.remote(y_future)
z = ray.get(z_future)
```

Ray Data

Activity

- Continue running cells in the "Loading large data with Ray Data" section
- This requires the Dataset we mounted at the beginning of the session
- Pay particular attention to the **Plasma** column in the Ray Web UI
- Stop when you get to the "Loading large data with Modin" section

Loading and manipulating data with Ray Data

- Ray Data allows reading files directly onto cluster workers
- Read parallelism is limited to the number of individual parquet files being read.
- Reading data takes several times the memory as the size on disk.
- Reading data is partially **lazy** - only the first file will be read until more data is needed
- Manipulating data with user-defined functions can be done two ways
  - Row-by-row with **map**
  - Vectorized operations on batches with **map_batches** (usually much faster)
- Data can **spill to disk** when needed

Ray Data is not intended for general ETL, but for last-mile processing and efficient loading into Ray.

Modin

Activity

- Continue running cells in the “Loading large data with Modin” section
- This requires the Dataset we mounted at the beginning of the session
- Pay particular attention to the **Plasma** column in the Ray Web UI
- Restart the workspace and skip to the Modin section to see it successfully execute parallel read on a larger file
- Stop when you get to the “Common Pitfalls for Large Data” section

Loading and manipulating data with Modin

- Modin allows reading data onto Ray clusters with a pandas interface.
- Modin will automatically break up large data, even from a single large file.
- Memory errors can occur when reading large single files AND cluster memory is already substantially in use.
- Manipulating data is done with the standard pandas API
    - The goal is to simply “import modin.pandas as pd” and use exactly as pandas
    - There is not yet 100% coverage of the pandas API

Modin is intended to be a seamless drop-in replacement for pandas for data at any scale.

Tips for Large Data and Memory

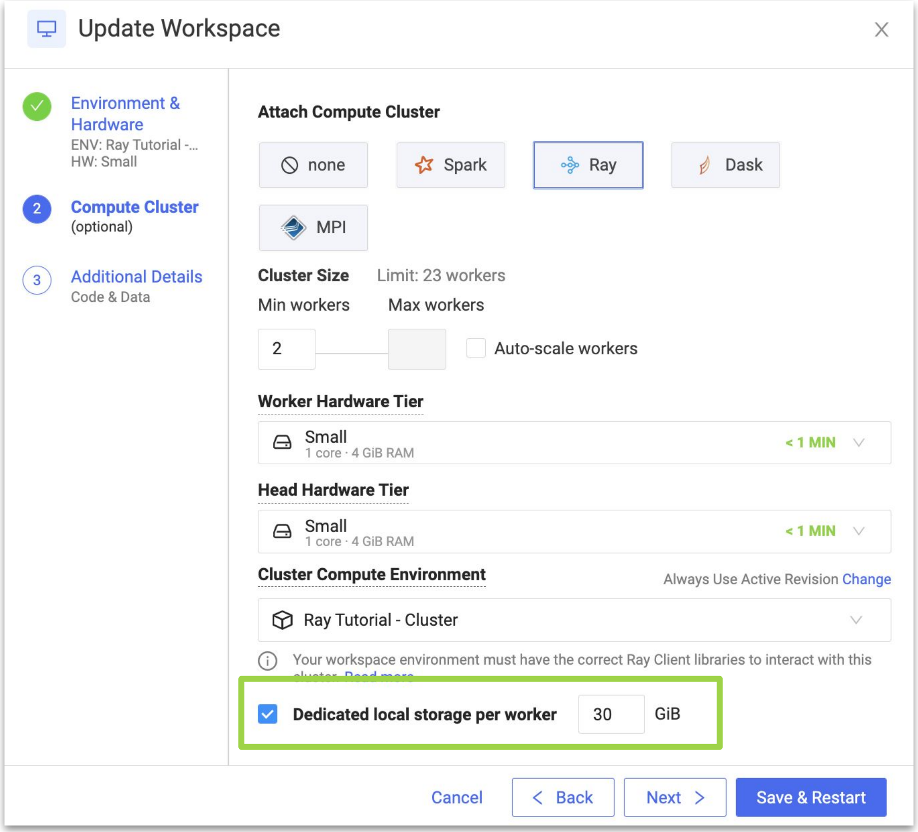
Tips for avoiding common pitfalls with large data

- Use ray.put and delay using ray.get where appropriate
- Remember that Remote tasks and actors cannot access files in /mnt, but they can access files mounted via /domino/datasets/, so store large files in Datasets.
- Always watch the Ray Web UI!
- Ray Data can only parallelize data read up to the number of individual parquet files being read, and it usually requires several times the on-disk size in memory.
- Modin can parallelize data read even for single large files, but is not always robust to a "messy" cluster when doing so.
- Any time a large data operation kills cluster workers and causes an error, beware the fact that it also likely kills the cluster connection! When in doubt restart the workspace entirely.

Bonus: dedicated local storage

Activity

- Run the cells in the final Bonus section
- Restart the Workspace with dedicated local storage enabled and run the cells again, comparing the difference.
- Note that object spilling is always possible, but the performance improves with dedicated local storage



linkedin.com/in/nikolaymanchev

twitter.com/nikolaymanchev

DOMINO

Thank You!