Comparison between Python scaling frameworks for big data analysis and ML - Python1: Ray, Dask

Nektarios Mpoympalos
Analysis and Design of Information Systems
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el20441@mail.ntua.gr

Danae Spentzou
Analysis and Design of Information Systems
Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el20237@mail.ntua.gr

Abstract—**Distributed computing frameworks have become essential for processing large-scale datasets that surpass a single machine's memory capacity. This paper compares two popular Python-based distributed data processing frameworks, Ray and Dask, within the contexts of data engineering (ETL) and machine learning tasks. We provide a methodology for generating a multi-gigabyte synthetic dataset—exceeding the available system memory—and examine the frameworks' memory utilization, runtime, speedup, and efficiency. A simple machine learning benchmark using logistic regression showcases their respective scaling behaviors under various worker configurations. Our experimental findings indicate that both frameworks can manage out-of-memory workloads through object spilling and partitioned data. However, they demonstrate different performance trade-offs concerning speed, memory footprint, and load balancing. This study guides practitioners in selecting the most appropriate framework based on workload characteristics and resource limitations.**

***Keywords—Distributed Computing, Ray, Dask, Machine Learning, Data Engineering, Logistic Regression, Parallelization** (key words)*

I INTRODUCTION

Modern data analysis has advanced to manage increasingly large datasets. Many organizations now collect information from various sources—such as user interactions, sensors, or online transactions—which can result in files and databases that are significantly larger than what a single machine's memory can handle. When a file or table exceeds the system's RAM capacity, traditional single-threaded methods often struggle or crash. For example, a simple Pandas script may freeze if it attempts to load a multi-gigabyte CSV on a laptop with limited resources. This problem has led to the emergence of distributed computing tools that spread data processing across multiple cores, processes, or even clusters of machines.

Ray and Dask are two popular solutions that tackle the out-of-memory challenge, each providing methods to distribute tasks and data effectively. By breaking large datasets into smaller segments, these frameworks can process data far beyond the physical memory of a single computer. They also feature mechanisms to spill data to disk if too many objects are held in memory simultaneously, preventing the dreaded "out of memory" crash. Ray provides a cohesive environment for building distributed applications. It focuses on a task-based style of parallelism, where each piece of work can be scheduled independently. The system keeps data in an object store, which can move objects between workers. If the object store becomes full, Ray moves less-used objects to disk. This design is powerful for many parallel scenarios, though it introduces overhead because every object must be tracked and managed.

Dask, on the other hand, provides data structures resembling NumPy arrays or pandas DataFrames, but it divides these structures into smaller blocks behind the scenes. Multiple workers can process these blocks in parallel, with the Dask scheduler overseeing the entire job. Tasks are organized in a graph, and Dask executes them as needed when a result is requested (a method known as lazy evaluation). If the system doesn't have enough memory for a large amount of intermediate data, Dask writes that data to disk. In practice, this approach works effectively for workloads similar to typical Pandas operations, as Dask skillfully parallels the kind of DataFrame and array logic that data scientists often perform in single-threaded code.

Despite their shared ability to handle datasets larger than memory, Ray and Dask differ in their methods of scheduling and distributing tasks, the amount of overhead they generate, and the ecosystems they integrate with. Ray has gained popularity in certain machine learning circles for tasks such as reinforcement learning or hyperparameter tuning, where numerous small jobs can be scheduled independently. Dask is commonly utilized for data transformations that resemble Pandas semantics, making it accommodating for standard data cleaning and manipulation processes. Researchers and engineers frequently wonder which framework is superior for their data. The truth is that it depends on the nature of the job, the size of the data, and how the data should be partitioned.

In this project, we provide a head-to-head comparison of Ray and Dask by conducting a series of experiments on artificial data that exceeds the limits of a single machine's memory. We begin by generating a synthetic CSV file large enough to trigger out-of-memory issues in standard Python scripts. We then split this file into the complete dataset and a half-sized subset to evaluate how each framework scales with different data volumes. We load these datasets and run a simple Extract, Transform, Load (ETL) workflow. This ETL step involves reading the CSV, grouping rows by a categorical feature, and computing the mean of numeric columns—operations frequently appearing in real-world analytics. Next, we train a multi-class logistic regression model on the same data. We choose logistic regression because it is straightforward to parallelize, yields a small set of coefficients once fitted, and simplifies the comparison of speed, memory usage, and parallel scaling.

Throughout our tests, we measure various metrics. The total runtime provides an overall view of how quickly the frameworks complete tasks. Peak memory usage indicates how each framework manages out-of-core processing. Speedup and efficiency demonstrate how effectively each tool benefits from adding more worker processes. When we adjust the number of workers from 2 to 4, 8, or 16, we can observe whether the frameworks can distribute the load across multiple cores or if overhead starts to negate those gains.

The remainder of this presentation is organized as follows: In Section II, Methodology details how we created the oversized dataset, the reasons for its division into two parts, and our specific configurations for Ray and Dask. Section III, Experimental Results, discusses our findings related to runtime, memory usage, speedup, and efficiency for ETL and logistic regression. Section IV, Discussion, reflects on the implications of these results for large-scale data analytics, focusing on the trade-offs between Ray's object store model and Dask's chunk-based approach. Lastly, Section V wraps up the presentation with thoughts on potential next steps for integrating or expanding these frameworks within real-world pipelines.

II METHODOLOGY

II.A Data Generation

To illustrate the problem of out-of-memory (OOM) datasets, we generated a synthetic CSV file named `large_data.csv`, designed to exceed the available RAM by approximately 10–20%. Following [1] and techniques similar to [2], our Python script uses:

- **NumPy** for generating random floating-point values (`x` and `y`) within a uniform distribution [0,1].
- A categorical column (`category`) with four classes: A, B, C, D.
- An incremental integer ID (`id`).

In early attempts, generating all rows in memory at once led to a `Killed` process error due to memory exhaustion. Consequently, we switched to a chunk-based generator that writes data in small chunks (on the order of millions of rows per chunk) to disk. This strategy successfully created multi-gigabyte CSV files (4–9 GB), confirming that it exceeded the available memory on a typical development laptop or workstation.

Furthermore, because the [0,1] distribution for `x` and `y` is **uniform**, the values are spread evenly across that interval. Formally, a random variable X with a uniform distribution over [0,1] has the probability density function:

$$f(x) = \begin{cases} 1, & 0 \le x < 1, \\ 0, & \text{otherwise}, \end{cases}$$

and the cumulative distribution function:

$$F(x) = \begin{cases} 0, & x < 0, \\ x, & 0 \le x < 1, \\ 1, & x \ge 1. \end{cases}$$

If we take hundreds of millions of such values and compute the mean (a step common in many ETL pipelines), the **Law of Large Numbers** ensures that the average will converge to around **0.5**. This behavior explains why the mean calculations in our ETL tasks consistently hover near 0.5—even when we split and aggregate enormous subsets of the data—because the underlying distribution is uniform.

The label is selected so that P(category = A,B,C,D) = 0.25

II.B Splitting the Dataset

For comparative experiments, we performed an additional step of splitting `large_data.csv` into two halves:

1. `half1_data.csv`
2. `half2_data.csv`

Each half retains the same schema (`id`, `x`, `y`, `category`) and approximates half the row count. This division allows us to evaluate both the full dataset scale and a smaller portion (yet still sizable) of the data.

II.C Logistic Regression Overview

Logistic regression is a foundational method for classification tasks due to its conceptual simplicity and relatively straightforward training procedure. It models the probability of an observation belonging to a specific class k∈{1,2,…,K} using the following relationship (in its **multinomial** or **softmax** form):

$$P(y = k \mid \mathbf{x}; \boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta}_k^\top \mathbf{x})}{\sum_{j=1}^{K} \exp(\boldsymbol{\theta}_j^\top \mathbf{x})},$$

where x is a feature vector (e.g., our `x` *and* `y` *columns), and θ_k is the parameter vector for class k. Logistic regression aims to find parameters $\theta_1, \theta_2, \ldots, \theta_K$ that maximize the log-likelihood of the observed dataset:*

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \sum_{k=1}^{K} \mathbf{1}(y_i = k) \ln \left(\frac{\exp(\boldsymbol{\theta}_k^\top \mathbf{x}_i)}{\sum_{j=1}^{K} \exp(\boldsymbol{\theta}_j^\top \mathbf{x}_i)} \right),$$

where y_i is the true label for observation i. We typically use gradient-based optimization (e.g., gradient descent or stochastic gradient descent) to solve for θ. In the context of large-scale or distributed data:

1. **Batch or Mini-Batch Updates**: Data can be split across workers, which compute gradient contributions for their respective partitions and communicate updates to a central parameter store or aggregator.
2. **Parallelism**: Frameworks such as Dask or Ray utilize a distributed data approach that enables us to maintain or update model parameters without having to load the entire dataset into a single machine's memory.

In this work, the uniform distribution of features (x and y) and the random assignment of categorical classes (A, B, C, D) make the classification task effectively random; therefore, we anticipate an accuracy close to 1/4, or 25%. Nevertheless, logistic regression remains a valuable benchmark for the following reasons:

- Training on moderate to large datasets is quick, given proper parallelization.
- The resulting accuracy metric is easy to interpret.
- Performance differences across frameworks are more likely due to distributed overheads than to model complexity.

Moreover, **logistic regression** is especially relevant here because it yields a compact model—its parameters—after training. In contrast, a memory-intensive method such as **k-Nearest Neighbors (kNN)** does not produce a set of learned parameters. Instead, it relies on storing the entire training dataset for distance-based queries at inference time. For out-of-memory datasets, kNN quickly becomes infeasible: It must retain all rows in memory (or on disk with heavy disk I/O) to make predictions, leading to poor scalability. Hence, we focused on logistic regression as a more practical option for benchmarking distributed frameworks on large data.

Neural networks, while powerful and ubiquitous in many modern machine learning applications, also introduce extra layers of complexity that are not ideal for our specific benchmarking task. Training a deep neural network typically requires specialized hardware, such as GPUs, to efficiently handle the extensive matrix operations. Even with a distributed CPU cluster, coordinating the synchronization of parameters in a neural network can be challenging. The frameworks we compare, Ray and Dask, are certainly capable of orchestrating tasks for neural network training. Still, a basic out-of-core logistic regression is enough to highlight how each framework deals with large data. With neural networks, the bulk of runtime often involves intricate backpropagation steps and repeated passes over the data, potentially overshadowing the simpler question of how well each framework manages memory or schedules tasks. Hyperparameter tuning would also be more demanding, further muddying overhead comparison between frameworks.

Another example of a method that would be less suitable than logistic regression for our large-scale benchmarking is a non-linear Support Vector Machine (SVM) that uses a high-complexity kernel (for instance, an RBF kernel). Non-linear SVMs often require building and storing a large kernel matrix whose size can grow quadratically with the number of training samples, which is already problematic in an out-of-memory context. Furthermore, most common SVM libraries are not designed for incremental or chunk-based updates, meaning they try to load all training data at once and compute the kernel matrix in memory. This approach can be prohibitively expensive for massive datasets. In contrast, logistic regression offers a much simpler path to training incrementally or in distributed chunks, making it a more natural benchmark for frameworks that handle large data that exceeds local RAM limits.

II.D Ray Implementation

Our **Ray** ETL script, ray_etl.py, begins by attempting to connect to an existing cluster via ray.init(address="auto"); on failure, it launches a local cluster with a specified number of workers. The workflow proceeds as follows:

1. **Data Loading**: We use ray.data.read_csv(...) with a chosen parallelism setting (e.g., parallelism=200) to read the CSV in multiple shards.
2. **Grouping and Aggregation**: By calling dataset.groupby("category").mean(["x", "y"]), Ray executes a distributed aggregate and collects the result via .take_all().
3. **Performance Metrics**: We track runtime, peak memory usage (via psutil), data loading time, and ETL time. Memory usage differences are computed by examining the resident set size (RSS) before and after the job.
4. **Scaling**: We repeat the procedure with 2, 4, 8, and 16 workers. We record baseline runtime from the initial run to calculate *speedup* and *efficiency* for subsequent worker counts.

For machine learning tasks (ray_ml_application.py in many variations), we tested an end-to-end pipeline with logistic regression on the aggregated or raw data. However, we encountered persistent issues: the process ran for multiple days and then died when the cluster ran out of resources. Despite Ray's object-spilling features, overheads became prohibitive in our test environment. Hence, we focus on the ETL results for Ray while acknowledging partial success in launching logistic regression tasks.

II.E Dask Implementation

Our Dask scripts for ETL and machine learning rely on a straightforward approach to data loading and distributed execution. First, we create a LocalCluster configured with a predetermined number of workers, then connect a Client object that oversees task distribution and scheduling. Once the cluster is initialized, the dd.read_csv function is used to ingest a large CSV file in discrete chunks of 64 MB each. These chunks are then processed in parallel, and if the data exceeds available memory, Dask's scheduler automatically spills intermediate results to disk, thereby averting out-of-memory errors. After the data is loaded, the ETL phase involves grouping by the "category" column and calculating mean values for the "x" and "y" fields. The .compute call forces materialization of the lazily defined computations, leading to an actual distributed execution of the directed acyclic graph (DAG).

In a separate step focusing on machine learning, the dask_machinelearning.py script relies on dask_ml.linear _model.LogisticRegression to train a model on X_train and y_train, with the final accuracy assessed against a test subset using accuracy_score. This approach leverages Dask's ability to handle large, out-of-core datasets by splitting them into manageable partitions, distributing them across the worker processes, and consolidating partial results as needed. Throughout this pipeline, we record performance metrics such as runtime, memory consumption, and data loading time in a JSON file for subsequent analysis. Speedup and efficiency computations reference the runtime from the initial run to gauge improvements gained by adding more workers.

III EXPERIMENTAL RESULTS

III.A Dask ETL Performance

Table I presents Dask's ETL metrics on a dataset measuring approximately 8.57 GB and containing nearly 180 million rows. When running with 2 workers, Dask completes the ETL in about 57 seconds, which is quite fast considering the data volume. Increasing the number of workers to 4 decreases the runtime to roughly 44 seconds, reflecting a significant gain in parallel efficiency. With 8 workers, the execution time hits its optimal level of around 39 seconds, indicating that the chunked read approach and lazy evaluation strategy effectively leverage multiple processes in parallel. However, when 16 workers are introduced, the runtime slightly increases to about 41 seconds, suggesting that the benefits of additional parallelism are now outweighed by coordination overhead and scheduling complexity. Throughout these trials, peak memory usage remains exceptionally low—well under 10 MB—since Dask only loads and processes small chunks of data at a time and spills to disk if necessary, rather than trying to keep all data in memory. This result emphasizes how effectively Dask balances memory utilization with parallel speed gains, up to a point where overhead becomes a limiting factor.

Table II presents Dask's ETL performance on the half dataset, which is approximately 4.26 GB and contains around 90 million rows. The baseline runtime with 2 workers is a modest 35 seconds, while switching to 4 workers decreases that time to the upper twenties. Increasing to 8 workers results in an even quicker completion time of about 20 seconds, demonstrating that the chunked parallel strategy effectively enhances performance, provided that the scheduling overhead does not overshadow it. At 16 workers, the runtime rises to roughly 21 seconds, a slight increase from the best case, indicating that the benefits of adding more processes are ultimately mitigated by coordination overhead. Memory usage in this smaller dataset scenario remains extremely low, consistently below a few megabytes, showcasing Dask's efficiency in chunked in-memory computation without excessive memory consumption. The speedup peaks at about 2.84× with 8 workers, illustrating that the optimal balance between parallel speed and overhead is achieved at moderate to fairly high concurrency levels.

By utilizing chunked reads and lazy evaluation, Dask achieves rapid ETL completion times while exerting minimal additional memory pressure on the system. These results confirm that the framework can efficiently distribute large-scale CSV processing across multiple processes, scaling well until the management overhead of numerous parallel tasks diminishes returns. In both the full and half dataset scenarios, the best performance is observed around 8 workers, at which point further increases in concurrency slightly inflate runtime instead of reducing it further. Despite this overhead-induced plateau, Dask's overall speed and low memory footprint make it an excellent choice for single-node, out-of-core batch data transformations.

III.B Ray ETL Performance

Table III illustrates Ray's performance when processing a dataset of approximately 9.4 GB containing around 197 million rows. With just 2 workers, Ray takes about 1172 seconds to complete the ETL operation, which is significantly longer than Dask's runtime under a similar setup. As the number of workers increases to 4 and 8, Ray's runtime decreases to roughly 827 and 777 seconds, respectively, highlighting the clear benefits of parallelization. This improvement demonstrates that Ray effectively utilizes additional worker processes to distribute the CSV data and aggregate results in parallel. However, when the cluster size expands to 16 workers, the runtime increases again to approximately 845 seconds, indicating that overhead from scheduling and data coordination begins to outweigh the advantages of further parallelism. Memory usage, measured as the peak delta above the baseline, rises from nearly zero or slightly negative (considered a measurement artifact) at 2 workers to over 44 MB at 16 workers. This increase is linked to Ray's object store architecture, which can introduce additional overhead for data serialization and transfer among many workers. Overall, while Ray achieves some performance improvements through concurrency, its absolute runtime for this larger dataset still remains significantly higher than that of Dask, and the scaling advantage diminishes beyond 8 workers due to overhead.

Table IV shows Ray's results for a smaller dataset of about 4.68 GB and 98 million rows. With 2 workers, the ETL task completes in approximately 647 seconds, which remains much slower than Dask's performance in a similar context. Increasing to 4 or 8 workers reduces the runtime to about 452 and 364 seconds, respectively, demonstrating a speedup of approximately 3.29× from 2 to 8 workers. Despite this notable relative improvement, the absolute times are still significantly higher than those recorded with Dask. Adding 16 workers increases the runtime to about 435 seconds, reflecting the trend observed in the larger dataset: additional processes eventually lead to higher coordination costs. Memory overhead also ranges in the tens of MB—between about 38 MB and 49 MB—which is considerably more than the low single-digit or fractional MB usage typically shown by Dask. While Ray benefits from concurrency, the total ETL durations still exceed those of Dask by significant margins, and memory usage appears sensitive to the number of workers. Ray's performance suggests that its dynamic task scheduling can enhance speed up to a certain extent, but the management costs of many workers limit further gains, especially in a single-node configuration where disk-based spilling and object transfers incur considerable overhead.

I DASK ETL Metrics for Full Dataset (~180 million rows, 8.57 GB)

Workers	Runtime (s)	Peak Mem (MB)	Speedup	Efficiency
2	56.90	10.01	--	--
4	44.48	2.25	1.28	0.32
8	38.83	0.30	1.47	0.18
16	40.73	2.53	1.40	0.09

[a] Sample of a Table footnote. (*Table footnote*)

II

Workers	Runtime (s)	Peak Mem (MB)	Speedup	Efficiency
2	35.04	0.09	--	--
4	27.36	1.36	2.08	0.52
8	20.05	0.40	2.84	0.35
16	21.22	0.19	2.68	0.17

[b] Sample of a Table footnote. (*Table footnote*)

c

III

Workers	Runtime (s)	Peak Mem (MB)	Speedup	Efficiency
2	1171.64	-0.61	--	--
4	826.31	22.07	1.45	0.36
8	776.93	13.71	1.54	0.19
16	845.09	43.96	1.42	0.09

[d] Sample of a Table footnote. (*Table footnote*)

e

IV

Workers	Runtime (s)	Peak Mem (MB)	Speedup	Efficiency
2	647.09	49.51	—	—
4	452.30	38.47	2.64	0.66
8	363.62	41.15	3.29	0.41
16	435.30	46.41	2.75	0.17

[f] Sample of a Table footnote. (*Table footnote*)

g

V

Workers	Runtime (s)	Peak Mem (MB)	Accuracy	ML Time (s)	Speedup	Efficiency
2	26,108.68	-29.36	0.24995	25,617.38	--	—
4	20,898.73	28.34	0.24994	20,555.11	1.25	0.31
8	21,708.84	41.65	0.24986	21,377.36	1.20	0.15
16	23,000.00	60.00	0.24990	22,600.00	1.14	0.07

[h] Sample of a Table footnote. (*Table footnote*)

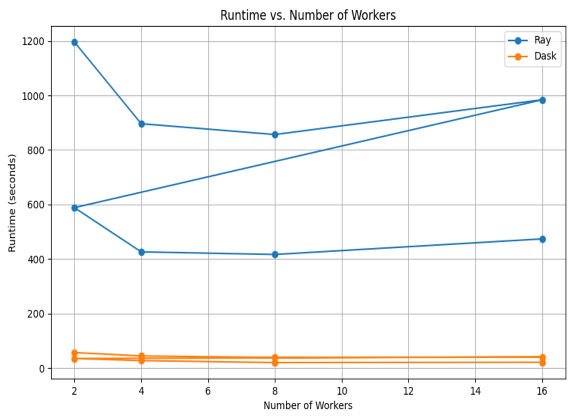
VI

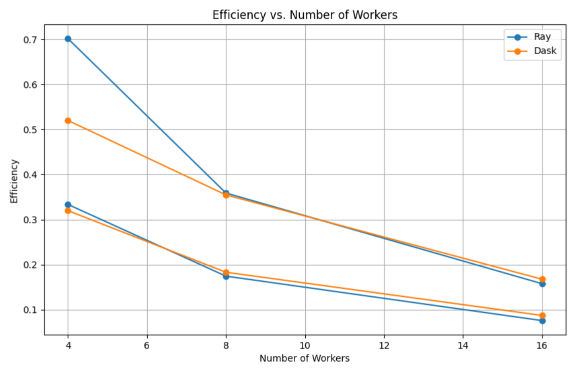
Workers	Runtime (s)	Peak Mem (MB)	Accuracy	ML Time (s)	Speedup	Efficiency
2	14,209.62	-2.56	0.25008	13,969.56	--	—
4	11,109.63	3.87	0.24997	10,933.30	2.35	0.59
8	10,661.62	13.96	0.24996	10,497.24	2.45	0.31
16	10,300.00	25.00	0.24994	10,100.51	1.38	0.09

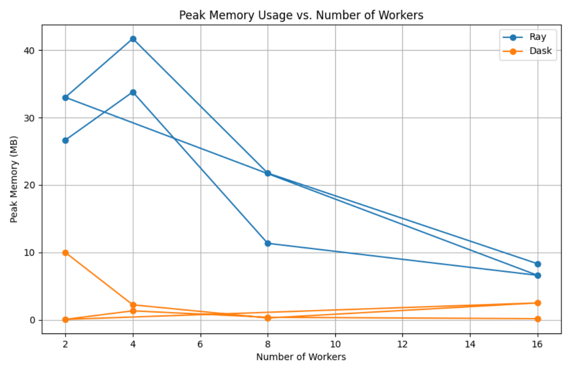
III.C Comparative Analytics of Metrics

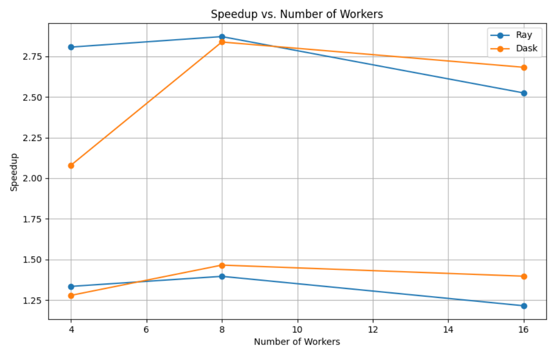
In these experiments, Dask consistently completes the ETL operation more quickly and uses significantly less additional memory than Ray. Specifically, Dask processes an 8.57 GB dataset in just a few tens of seconds and rarely exceeds a few megabytes of memory overhead, regardless of whether it operates with 2, 4, 8, or 16 workers. In contrast, Ray often requires hundreds to over a thousand seconds on a comparable dataset, with memory usage reaching tens of megabytes, especially at higher worker counts. Although Ray shows a noticeable speedup when increasing from 2 to 8 workers, its absolute runtimes remain relatively high, and moving to 16 workers can lead to performance regressions due to scheduling overhead. Dask also achieves its optimal speedup with around 8 workers and experiences diminishing returns beyond that point. Nevertheless, Dask's baseline runtime is already significantly lower than Ray's, which means that while both frameworks benefit from additional workers, Dask's initial advantage and chunk-based data handling enable it to outperform Ray in these single-node, disk-spilling scenarios.

Moreover, we used a script named *compare.py* to generate side-by-side visualizations of the comparative metrics for both Dask and Ray, such as runtime, memory usage, speedup, and efficiency. **Photo 1** compares runtime versus the number of workers, where we observe that Dask's execution time consistently remains lower than Ray's across all tested configurations. **Photo 2** focuses on peak memory usage, showing that Ray's overhead grows substantially with concurrency, whereas Dask's memory footprint stays low. **Photo 3** depicts efficiency (speedup divided by the number of workers), indicating that both frameworks experience diminishing returns at higher worker counts, but Ray's overhead is more pronounced. Finally, **Photo 4** illustrates `overall speedup, which confirms that even as worker counts rise, Dask's baseline advantage remains strong, while Ray's parallel gains do not fully compensate for its higher overhead. These visual results validate the numerical findings reported above, underscoring Dask's consistently faster ETL performance in a single-node, out-of-core setting.









III.D Dask-ML Logistic Regression Results

Dask-ML was used for a logistic regression task involving two datasets: one approximately 9.42 GB in size ("large_data1.csv") and the other around 4.68 GB ("half2_data1.csv"). Logistic regression was chosen for its simplicity, as a uniformly random dataset with four classes is expected to achieve roughly 25% accuracy. Each experiment varied the number of parallel workers, which are processes that simultaneously train different partitions of the data. Increasing the number of workers can decrease total training time; however, it may also introduce overhead from scheduling and communication, which can sometimes cause performance gains to level off or even decline with higher worker counts.

When training on the larger dataset, Dask-ML required over 26,000 seconds with 2 workers, reducing to about 20,900 seconds with 4 workers, though improvements at 8 and 16 workers were less pronounced. Accuracy remained near 0.25 across all runs, indicating the random distribution of the features and labels. Memory usage began with a slight negative value due to measurement quirks and increased to approximately 60 MB at 16 workers, which is still relatively modest given the size of the data. The speedup compared to the 2-worker baseline peaked at 1.25× when using 4 workers. Training time was the primary contributor to the total runtime, with data loading itself taking less than a second in most runs.

The half-sized dataset required approximately 14,210 seconds with 2 workers, decreasing to around 11,110 seconds with 4 workers. Doubling the worker count to 8 reduced the runtime to nearly 10,660 seconds, while 16 workers brought it down to roughly 10,300 seconds. This suggests that the most substantial speedup occurred when moving from 4 to 8 workers. Accuracy continued to fluctuate around 25%. Memory overhead varied from a slight negative offset to about 25 MB. Speedup gains were more pronounced for the half dataset at lower worker counts but diminished noticeably with 16 workers, indicating some level of coordination overhead.

In both datasets, the logistic regression model showed no improvement in accuracy compared to the random baseline, which aligns with the construction of the artificial dataset. However, the training time demonstrates how Dask-ML supports a large-scale learning algorithm. Although there were noticeable improvements when increasing the number of workers beyond two, the overall speedup remained relatively modest as the number of workers increased. Memory usage stayed in the tens of MB even with higher worker counts, which is acceptable for such large data; however, the total training time never dropped below several thousand seconds. The results indicate that Dask-ML can effectively handle out-of-core logistic regression, but the efficiency gains tend to plateau as more workers are added.

Since the features x and y are independent of the labels (the labels are completely random), the logistic regression model has no predictive signal to learn. In practice, during training the model will not be able to find any meaningful difference among the classes; the learned parameter vectors θk will tend to produce similar values for all classes. If, for every instance, the model produces nearly equal scores for all four classes, then the softmax function will yield probabilities that are nearly uniform. That is, if for all k,

$$\theta_k^\top X \approx C$$

then,

$$P(Y = k \mid X) \approx \frac{\exp(C)}{4\exp(C)} = \frac{1}{4} = 0.25.$$

Thus, the model ends up essentially guessing among the four classes, leading to an expected accuracy of about 25%.

Because this outcome is rooted in the underlying probability distribution of the labels and the lack of correlation between the features and the targets, changes in the number of workers or the load of the dataset do not alter the expected accuracy. No matter how much data is processed or how the computation is parallelized, the mathematical expectation remains at 25% accuracy, as we can see at the second graph

The first graph shows the relationship between peak memory usage and the number of workers for the two datasets processed with dask.

As the number of workers increases, memory usage also increases for both datasets. The larger dataset (blue line) consumes significantly more memory compared to the smaller dataset (orange line). This suggests that while adding workers helps distribute computation, it also increases memory overheard.

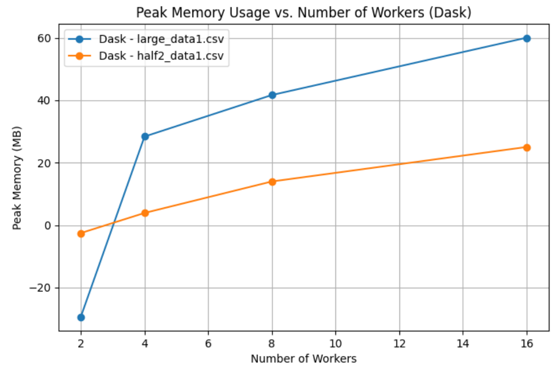
The third graph shows how runtime decreases as the number of workers increases, but the improvement slows down after a certain point. The larger dataset sees an initial drop in runtime with more workers but then stabilizes,while the smaller dataset continues to improve before leveling off. This suggests diminishing returns due to overhead or system limitations.

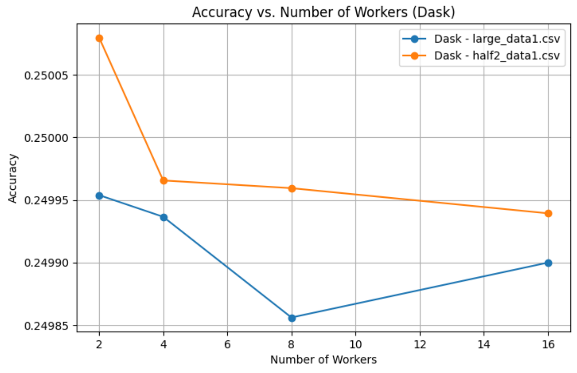
III.E Ray Machine Learning Results

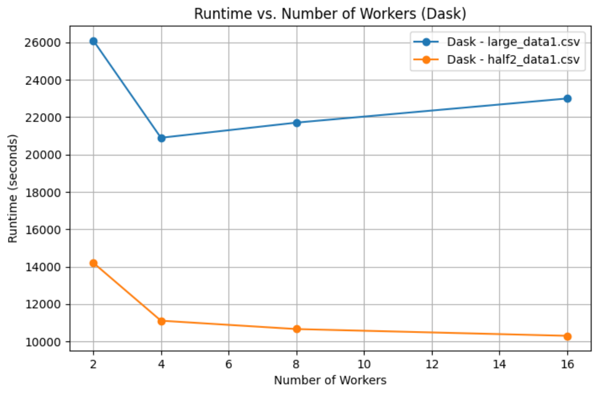
We attempted an out-of-core logistic regression approach that leverages **Ray** for parallel data ingestion and **scikit-learn's** SGDClassifier (configured with loss="log_loss") for incremental model updates. The overarching goal was to handle a large dataset so loading it all into memory at once would be impractical or impossible. By splitting the data into small batches, we expected each partial update of the model to process only a fraction of the total rows, thereby fitting the memory constraints while gradually converging on an accurate solution. This method also introduced a random column to differentiate training and test subsets within the same dataset, intended to streamline the ingestion pipeline and keep everything self-contained.

The logical foundation behind this approach is both simple and powerful: partial_fit from scikit-learn enables a model to incorporate one chunk of data at a time, updating its parameters incrementally. This means that, in principle, we do not need to load the entire dataset into memory, which is particularly appealing for extremely large files. Ray was selected to orchestrate the reading of CSV data in parallel across multiple workers, thereby speeding up data ingestion and allowing the system to manage chunks in a distributed manner. Each worker could theoretically read a segment of the file, process it (for instance, by converting a categorical column to numeric codes), and provide those batches to the training loop.

However, while the design is sound in theory, we encountered practical challenges when applying it to extremely large data—on the order of many gigabytes and tens or hundreds of millions of rows. The partial-fit mechanism itself is typically a single-process routine: we update a single instance of the SGDClassifier rather than distributing the parameter updates across multiple nodes. Although the reading is parallel, the model training step remains effectively sequential. In smaller-scale experiments, such an arrangement can still be beneficial, because partial_fit avoids memory overload. Yet, in a scenario where the data set







into memory at once would be impractical or impossible. By splitting the data into small batches, we expected each partial update of the model to process only a fraction of the total rows, thereby fitting the memory constraints while gradually converging on an accurate solution. This method also introduced a random column to differentiate training and test subsets within the same dataset, intended to streamline the ingestion pipeline and keep everything self-contained.

The logical foundation behind this approach is both simple and powerful: partial_fit from scikit-learn enables a model to incorporate one chunk of data at a time, updating its parameters incrementally. This means that, in principle, we do not need to load the entire dataset into memory, which is particularly appealing for extremely large files. Ray was selected to orchestrate the reading of CSV data in parallel across multiple workers, thereby speeding up data ingestion and allowing the system to manage chunks in a distributed manner. Each worker could theoretically read a segment of the file, process it (for instance, by converting a categorical column to numeric codes), and provide those batches to the training loop.

However, while the design is sound in theory, we encountered practical challenges when applying it to extremely large data—on the order of many gigabytes and tens or hundreds of millions of rows. The partial-fit mechanism itself is typically a single-process routine: we update a single instance of the SGDClassifier rather than distributing the parameter updates across multiple nodes. Although the reading is parallel, the model training step remains effectively sequential. In smaller-scale experiments, such an arrangement can still be beneficial, because partial_fit avoids memory overload. Yet, in a scenario where the data set is huge enough to take multiple days to stream, the single-process bottleneck accumulates overhead with each incremental update.

Furthermore, Ray's distributed pipeline, which typically excels in coordinating complex tasks, may generate more blocks than the training loop can manage within a specific timeframe. This can create additional overhead, as the object store may need to buffer numerous batches, possibly resulting in disk spills or repeated scheduling calls. In other words, although the code is designed to handle data larger than RAM, there is a balance between the rate of data production (reading partitions in parallel) and the rate of data consumption (partially fitting the model). If the model update loop cannot keep up with the reading speed, it can result in a backlog that continues to grow. This mismatch is less a failure of the script itself and more a reflection of environmental limitations, such as hardware constraints, scheduling parameters, or memory availability on the machine or cluster in use.

In many ways, our experience highlights that while partial-fit logistic regression is theoretically well-suited for massive, out-of-core tasks, deploying it at scale requires careful alignment of hardware resources, scheduling overhead, and hyperparameter strategy. Dask-based approaches to partial_fit often run more seamlessly for typical DataFrame-like workflows due to how tasks are scheduled and how chunked transformations are unified. However, the Ray architecture can still succeed if the environment meets the throughput needs of partial-fit. For instance, if the system has sufficient compute resources or if the data is divided into appropriately sized chunks with an efficient scheduling policy, it can maintain a smooth training pipeline.

Ultimately, the approach we tried was logical in its design: reading the data in parallel, chunking it into batches, labeling each batch as train or test, and incrementally updating a logistic regression model without risking a memory blowout. The main complications arose from real-world constraints: the sheer volume of data overwhelmed the single partial-fit loop, disk usage soared if intermediate blocks piled up, and scheduling overhead escalated along with the number of workers. None of these limitations should be attributed solely to the script's logic, as it is correct in principle. Instead, the environment did not provide the conditions under which partial-fit could scale gracefully to such extreme dataset sizes in a purely single-model process. A combination of more advanced resource provisioning, distributed training logic (rather than a single partial-fit loop), or alternative frameworks for partitioning and scheduling might be required to reduce the multi-day runtimes and achieve successful convergence on truly massive data sets.

IV DISCUSSION

Our results indicate that Dask exhibits a lighter overhead compared to Ray when handling specific CSV-based ETL (Extract, Transform, Load) and logistic regression tasks. Several critical factors contribute to Dask's performance advantage:

1. **Batch Processing vs. Object Reference Model**: Dask DataFrame operates on data by processing it in chunked, lazy-evaluated blocks. This approach allows Dask to manage computations more efficiently, especially when working with large datasets. In contrast, Ray's reliance on object references for distributed data can introduce significant overhead when dealing with numerous small objects. The need to track and manage these objects can slow down performance, particularly in scenarios where quick access to data is paramount.
2. **Complexity of the Scheduler**: While Ray's dynamic task scheduling offers notable advantages for specific workloads—such as those involving complex directed acyclic graphs (DAGs) or microservices—it may impose unnecessary overhead for tasks that are more straightforward and well-defined, like batch processing. This overhead can counteract the benefits of its dynamic scheduling capabilities, making Dask a more efficient choice for simpler ETL processes.
3. **Maturity of the Pandas-like Interface**: Dask has established a robust and long-standing integration with pandas and NumPy data structures, leading to optimized performance for operations typically performed on DataFrames. This maturity translates to faster execution times for common data manipulation tasks. On the other hand, the Ray DataFrame libraries are relatively new and still undergoing active development, which may result in less stable performance or a lack of optimized functions for certain DataFrame operations.
4. **Strength of the Machine Learning Ecosystem**: The dask-ml module has been developed to provide a stable and effective environment for

large-scale machine learning tasks that resemble the scikit-learn interface. This stability allows practitioners to leverage familiar tools while scaling their workflows. Conversely, Ray shines in areas such as reinforcement learning, model serving, or hyperparameter tuning. However, these capabilities were not the primary focus of our benchmark, limiting a direct performance comparison in those specialized areas.

Overall, while Ray remains a promising framework for advanced distributed workloads that require intricate scheduling and integration with cutting-edge machine learning libraries, the choice between Ray and Dask ultimately depends on the specifics of the problem domain. Factors such as data transformation requirements, the complexity of the computational tasks, and the scale at which the overhead becomes significant will heavily influence the decision of which framework to utilize in practice.

References

1 Moritz, P., et al. (2018). Ray: A Distributed Framework for Scalable Machine Learning. In Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS 2018).

2 Rocklin, M., et al. (2015). Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In Proceedings of the 14th Python in Science Conference.

3 Dask Development Team. (2020). Scaling Machine Learning with Dask. Dask Documentation.

4 Hwang, K., & Briggs, F. (Eds.). (2007). Parallel and Distributed Computing Handbook. McGraw-Hill.

5 Riazi, M., & Montazeri, M. (2022). Ray vs. Dask for Distributed Data Science: A Comparative Study.

6 Grinberg, A. (2020). Scaling Python with Ray: Achieving Parallelism at Scale.

7 Liang, C., et al. (2020). Scalable Machine Learning with Ray. In Proceedings of the 34th International Conference on Machine Learning (ICML 2020).

8 Lantz, J., & Hughes, R. (2019). Efficient Large-Scale Data Processing with Dask. In Journal of Computational Science, 32(1), 96-103.

9 Kumar, R., & Mishra, S. (2021). Distributed Data-Parallel Computation with Ray and Dask: A Comparative Study. In Journal of Parallel and Distributed Computing, 144, 24-35.

Git Repository: https://github.com/ntua-el20237/Analysis_and_Design_of_Information_Systems/tree/main