# Distributed Data Processing and Transformation Using Dask

## Overview

This project implements a distributed data processing pipeline using Dask to handle large datasets efficiently. It integrates custom data transformation techniques using Bayesian Gaussian Mixture Models (BGMM) for scaling and normalizing data. The key features include parallelized computation, advanced data transformation techniques, and scalability to process datasets up to 1 billion rows.

## Core Modules

### Main Pipeline (main.py)

- Configures Dask with 2 workers, each limited to 3GB memory, ensuring efficient resource management. This can be scaled to more workers as per the cpu resources available, Let’s say we have like 100GB Memory and 10CPUs resource allocated to our namespace in our cluster. We try to load the dataset the maximum it can accommodate within our limits then we process that much data with several workers in line.

For example 1 worker consume 1 Gb of memory which is equivalent to 1 million rows, then we with our 10 CPUs we can then run 100 workers totaling 100GB which runs 100million rows in one go. Next chunk will be processed once a worker gets free.   
  
Now if there is some preprocessing layer in between or we need to call some 3rd party API for data preprocessing like creating embedding vectors we can scale this efficiently using Celery and RabbitMQ for distributed task queuing.

If we have Spark Cluster we can run 1 billion rows at once where executors and nodes within the cluster efficiently distribute this into different executors and nodes to avoid OOM errors for larger than memory datasets. For a 100GB dataset the processing needs will be immense  
  
If the dataset or intermediate results cannot fit into memory, Spark writes the data to disk spill files as I have been doing the same with Dask library. If we’re not having that scale then Dask, Ray, Polars is sufficient as Spark is difficult to maintain further   
  
- Loads a data sample and fits the ScaleVGM transformer, which uses BGMM to cluster and normalize data.

According to the above settings we can do the same here as well that whatever the maximum we can fit into the data, we can understand the underlying distributions of the data and scale to billion rows accordingly to no miss the data distribution.

### Data Processing (data\_processor.py)

- Utilizes Dask to partition large datasets into manageable chunks.  
- Applies the ScaleVGM transformer to each partition for normalization and clustering.

### Scaling Dataset (distributed\_data\_scaling.py)

- Processes data chunks in parallel using Python’s multiprocessing library and Dask.

- Here I haven’t used Dask distributed properties because for this task, python’s multiprocessing with Dask partition worked faster than former as it was taking almost 150 to 200 seconds. Adding some noise and little complex calculations will take 300-400 seconds. Ultimately data scaled operation for 1 billion records has completed under 4-5 minutes.

**NOTE:** Here it is just a one column but if we have 100s of rows then I would go for Dask/Ray compute engines using multiprocessing

### Statistics and Reporting (stats.py)

Computes descriptive statistics such as mean, standard deviation, and missing values for numeric columns.  
Estimates memory usage for the dataset to optimize resource allocation.

## Dask Dashboard Insights

The provided Dask dashboard visualization gives the following insights:  
- Shows efficient parallel task execution with minimal task queuing, indicating optimized resource utilization.  
- Two workers are processing partitions with balanced workloads, leveraging Dask’s distributed computing capabilities.  
- The 'Bytes Stored' metric indicates the dataset size is within the allocated memory limits for the workers, preventing memory overflow.

This can be monitored on localhost:8787/status to monitor the live processing of the data. When we have these jobs running on Kubernetes, we can utilize the Grafana dashboard to monitor

## 4. Performance

- By processing partitions in parallel, the pipeline achieves significant time savings compared to sequential execution.  
- The system can handle extremely large datasets by splitting them into smaller chunks and using Dask’s disk spilling feature.  
- Customizable components (e.g., number of workers, memory limits, partition size) ensure adaptability to various system configurations.

## 5. Can be Improved

- Dynamic Worker Allocation: Using more workers dynamically based on system capabilities can improve processing speed for larger datasets. Which I had discussed at the start we can monitor using Grafana dashboards for the first time and from later cases we can Celery like distributed processing to effectively use no of workers which runs.  
  
If an 8GB computer process this within 12min for one column then a namespace with 512 GB RAM and with more features we need to use better resources though

I tried to use GPU level acceleration but could not find any part to activate a GPU in this case and may be if use neural network architecture to generate synthetic data, we can utilize multi-GPU training for these jobs which runs much faster.

1. Multi GPU (If NN architecture is used)
2. Dynamic Worker Allocation using Distributed Task Queuing
3. CICD Pipeline to trigger all the pipelines
4. Cron Job in Kubernetes to run the job daily
5. GPU level inference access for cluster if we need to write a Microservice to call the third-party API
6. Asynchronous option of FastAPI to accommodate more requests for an example if we use real time streaming services like Kafka
7. It works much faster if we load this data into databases and partitioned the dataset accordingly and process each partition into one memory.
   1. For example, some product related information data is different in different countries, so synthetic data generated should be better in terms of each country id will be close to real data I believe.