Deep Learning on Cloud:
Reducing cloud costs using the right approach

Scope

DeepIQ is a self-service {Data + AI} Ops app built for the industrial world. DeepIQ simplifies industrial analytics by automating the following three tasks:

1. ingesting operational and geospatial data at scale into your cloud platform.
2. implementing sophisticated time series and geospatial data engineering workflows; and
3. building state of the art ML models using these datasets.

Training deep learning models on large datasets is often very computationally intensive. If this process is not efficiently implemented, your cloud costs can increase significantly. In this whitepaper, we will focus on this topic and elaborate on how DeepIQ can help contain cloud costs when implementing complex, data-intensive workflows. We will use an example of a 3D Geospatial learning problem and compare the training times and costs with different execution strategies on Azure cloud using Databricks as the compute engine.

Background

Problem Specification

For brevity, we will provide only the details of this use case that are relevant to the topic of discussion. Readers interested in more details about DeepIQ's support for Deep Learning can reach out separately for more detail. We have chosen a binary classification problem that predicts if a certain 3D volume has a geospatial property of interest. We have a total of 650,000 volumes with each volume having cuboid of size (30 x 30 x 10) and 10 features. That is, each volume has 9,000 voxels, and for each voxel, we have 10 channels that can be used as features. We would like to associate a class label for each volume that indicates a geospatial property pertaining to that volume, such as expected permeability.

Experiment Design

For this whitepaper, we will train the same Deep Learning model using four different compute options and tabulate the training times with similar settings. These options are:

- single Node CPU Cluster using Azure Blob storage
- single Node CPU Cluster using Delta tables
- single Node GPU Cluster using Azure Blob storage
- single Node GPU Cluster using Delta tables

Why Storage I/O is important?

As is well known, GPU compute significantly improves deep learning model training. Additionally, ensuring that your I/O operations can scale to the compute capacity of your cloud machines will play an important role in making the training process efficient. Cloud storage can be significantly slower in reading a file when compared to local storage. As a result, for a significant fraction of the training time, your computer resources will remain idle. During this idle time, you will continue to be billed for expensive compute machines. One way to solve this problem is to load the whole dataset in memory before training. However, when the data sizes are large, this process is prohibitively expensive.

Alternatively, we can use optimized storage structures that provide faster I/0. For the Databricks platform, DeepIQ's prebuild components provide this functionality out of the box as per their recommended best practices. Databricks recommends the use of column-oriented parquet files to store data for deep learning use cases. DeepIQ has prebuilt components that can persist data into this optimized file format. DeepIQ's machine learning pipeline component leverages the Petastorm library along with the storage format improvements, to make this optimized storage compatible with deep learning libraries and provide significant improvement in throughput.

The Details

Cluster Creation

For this experiment, we created two different DeepIQ clusters - a single node CPU cluster and a single node GPU cluster. Within DeepIQ, you can create Databricks clusters so that all required libraries for our experiments are installed by default. The cluster configuration for both CPU and GPU clusters are shown in Figure 1 and Figure 2. For the CPU cluster, we choose a cluster with a 64 MB RAM since the experiments were significantly slower on smaller nodes.

Size	vCPU	Memory: GiB	Temp storage (SSD) GiB	GPU	GPU memory: GiB	Max data disks	Max NICs / Expected network bandwidth (Mbps)
Standard_NC4as_T4_v3	4	28	180	1	16	8	2 / 8000
Standard_NC8as_T4_v3	8	56	360	1	16	16	4 / 8000
Standard_NC16as_T4_v3	16	110	360	1	16	32	8 / 8000
Standard_NC64as_T4_v3	64	440	2880	4	64	32	8 / 32000

Figure 1:- GPU Machine Configuration

Size	vCPU	Memory: GiB	Temp storage (SSD) GiB	Max data disks	Max cached and temp storage throughput: IOPS/MBps (cache size in GiB)	Max burst cached and temp storage throughput: IOPS/MBps[2]	Max uncached disk throughput: IOPS/MBps	Max burst uncached disk throughput: IOPS/MBps[1]	Max NICs/ Expected network bandwidth (Mbps)
Standard_D2s_v3[2]	2	8	16	4	4000/32 (50)	4000/200	3200/48	4000/200	2/1000
Standard_D4s_v3	4	16	32	8	8000/64 (100)	8000/200	6400/96	8000/200	2/2000
Standard_D8s_v3	8	32	64	16	16000/128 (200)	16000/400	12800/192	16000/400	4/4000
Standard_D16s_v3	16	64	128	32	32000/256 (400)	32000/800	25600/384	32000/800	8/8000
Standard_D32s_v3	32	128	256	32	64000/512 (800)	64000/1600	51200/768	64000/1600	8/16000
Standard_D48s_v3	48	192	384	32	96000/768 (1200)	96000/2000	76800/1152	80000/2000	8/24000
Standard_D64s_v3	64	256	512	32	128000/1024 (1600)	128000/2000	80000/1200	80000/2000	8/30000

Figure 2: CPU Machine Configuration

Data Preparation

We transformed the 3D volume data in two different formats. For the first format, we used Azure Blob storage to save the data in a numpy array format. For the second format, we used a DeepIQ data workflow shown in Figure 3 to store these numpy arrays in a parquet format.

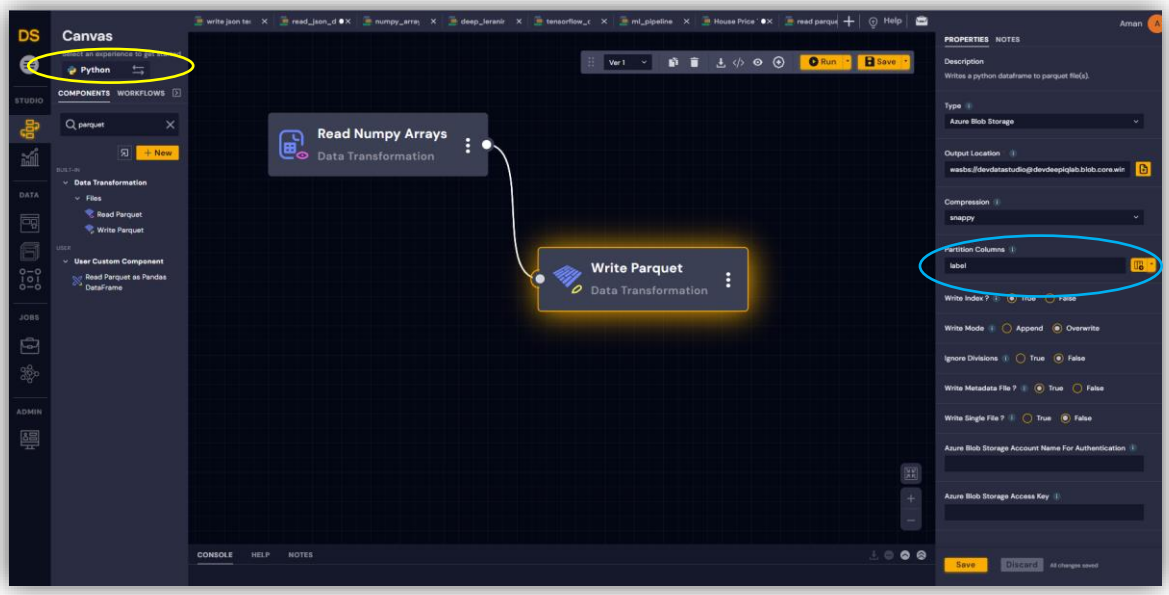

Figure 3: Saving Numpy Arrays in Parquet Format

To further improve I/O speed, we converted the parquet files to a single delta table so that meta-data corresponding to the data sets can be efficiently queried. As a last step, we optimized the delta table using an additional workflow shown in Figure 4 to reduce the number of parquet files associated with the delta table and improve I/O speed.

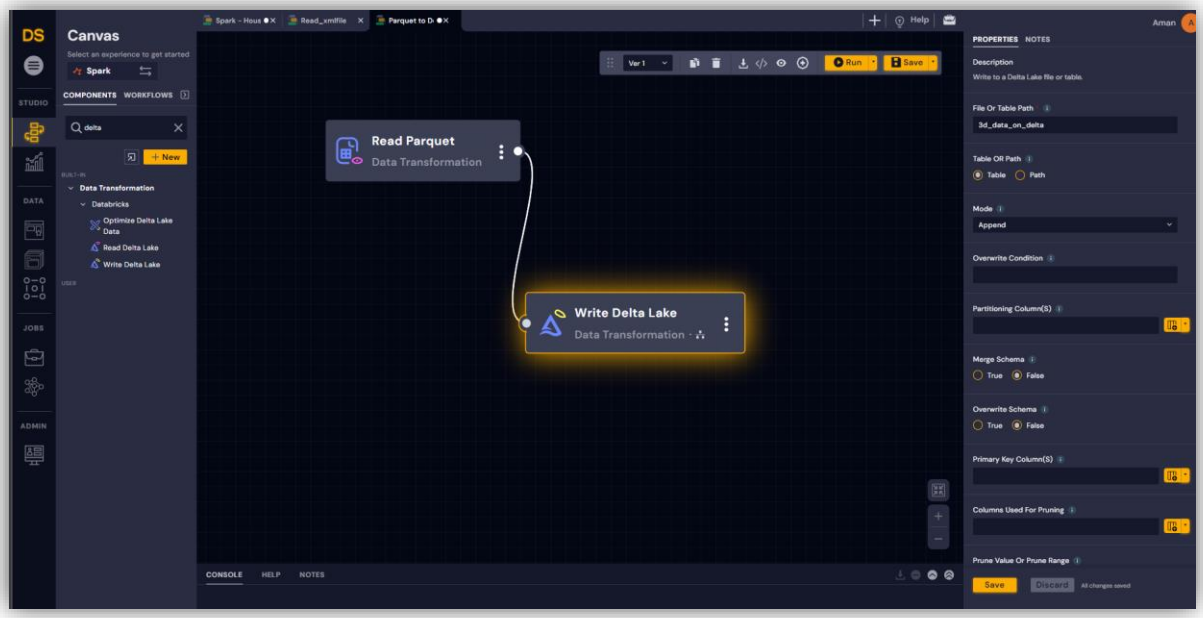

Figure 4: Using Parquet Files to Write to Delta Tables

The optimized delta table reduced the number of files from 650,000 to 3,000.

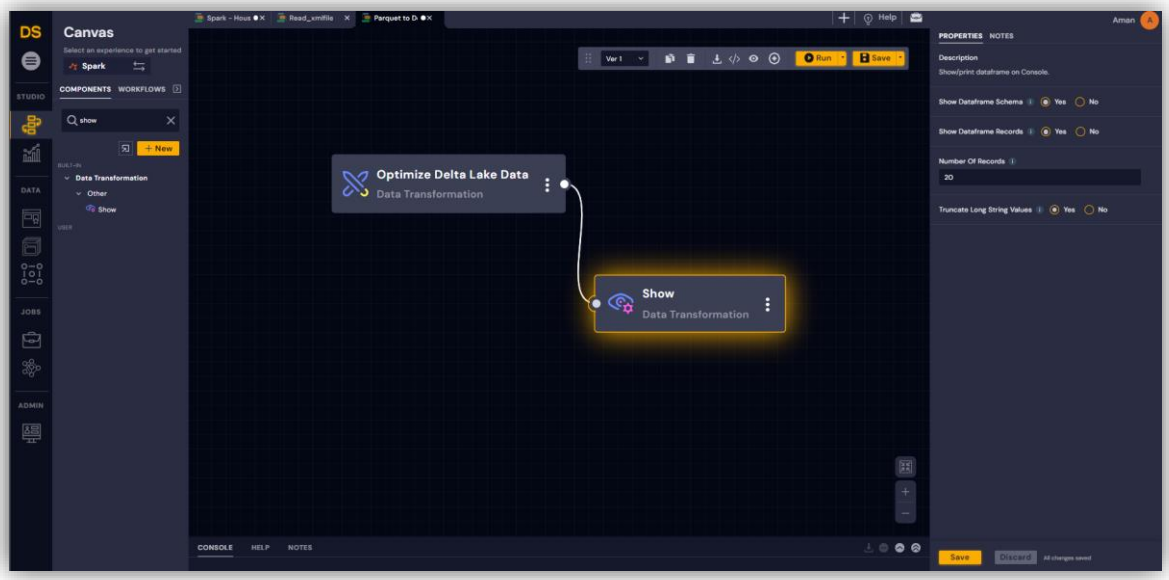

Figure 5: Optimizing Delta Tables

Machine Learning Model Training

Once we have the two types of clusters and two types of data ready, we used DeepIQ Machine Learning pipelines to train the model as shown in Figure 6. We trained this model for 5 epochs and with a batch size of 32, using 600 steps_per_epoch.

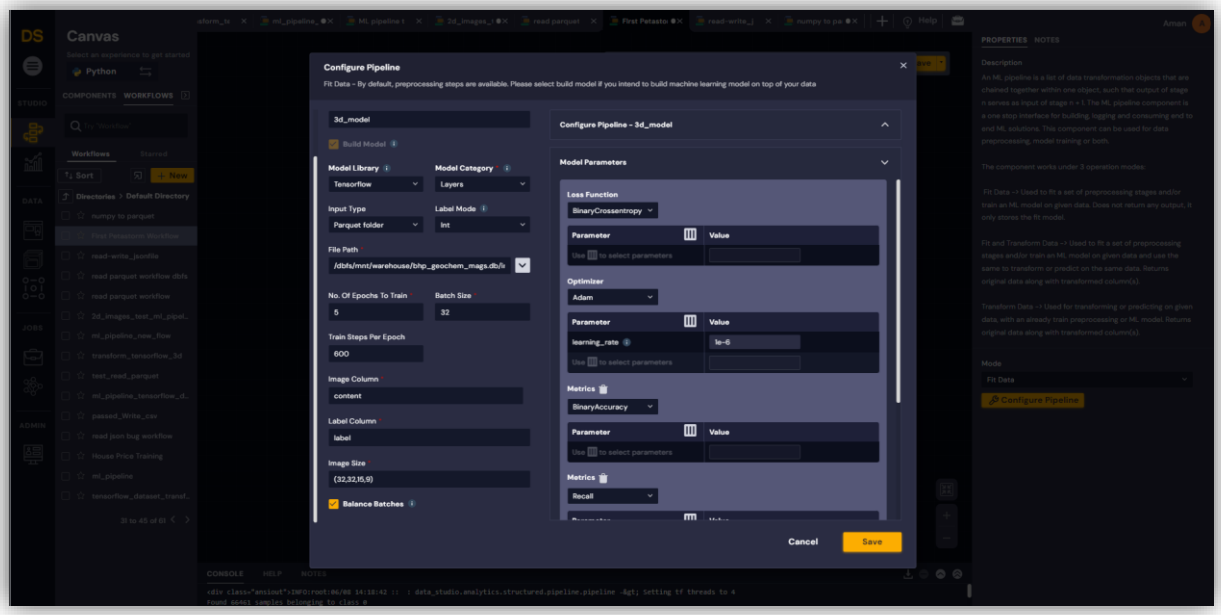

Figure 6: DeepIQ Machine Learning Pipeline UI

For the model, we used a ResNet backbone with the following structure.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
model (Functional)           (None, 1, 1, 1, 512)      33368384
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 513
=================================================================
Total params: 33,368,897
Trainable params: 33,361,089
Non-trainable params: 7,808
_________________________________________________________________
```

Figure 7 - Deep Learning Model Summary

As the model structure shows, the number of tuneable parameters is 33,361,089.

Results and Analysis

The final training times required to finish the training process are shown in Figure 7. As expected, the GPU machine generated significant efficiency gains by reducing the compute time by half, when using the simple cloud storage. Additionally, the I/O optimization paid significant dividends reducing the compute times by nearly 83% of what was observed using vanilla cloud storage.

Cluster	Optimized Disk I/O	Simple Cloud Storage
Single node GPU	19 minutes	1.8 hours
Single node CPU	2.42 hours	3.59 hours

Figure 7: Final Training Times

To understand the root cause of this result, let us deep us look at the Databricks Ganglia UI snapshot. We show three snapshots corresponding to three different experiments. When we ran the training model using Azure Blob as the storage on the single node, CPU cluster, I/O peaked at 7 Mbps (Figure 8). Migrating the data to the optimized delta lake structure increased the max I/O to 38 Mbps but as the bottom right I/O plot shows, CPU is now the limiting factor (Figure 9). Once we shift to the GPU based cluster (Figure 10), we were able to achieve a smooth I/O of 100 Mbps without any intermittent drops.

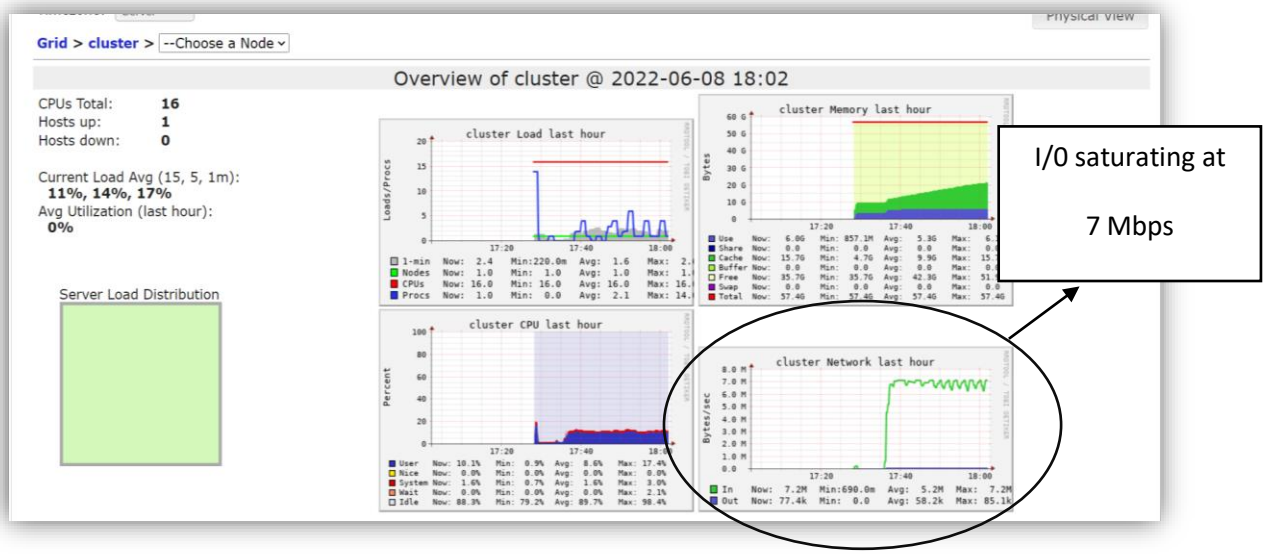

Figure 8: Training on CPU Machine with Simple Cloud Storage

.

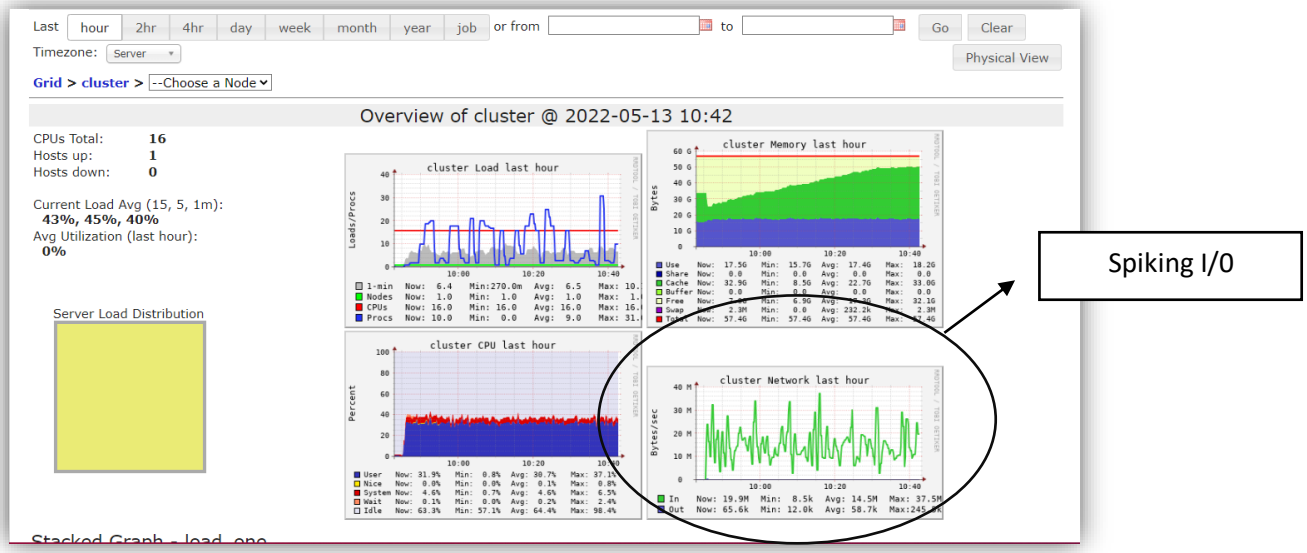

Figure 9: Training on CPU Machine with Optimized I/O

1880 Dairy Ashford, Suite 208A
Houston TX 77077
https://deepiq.com

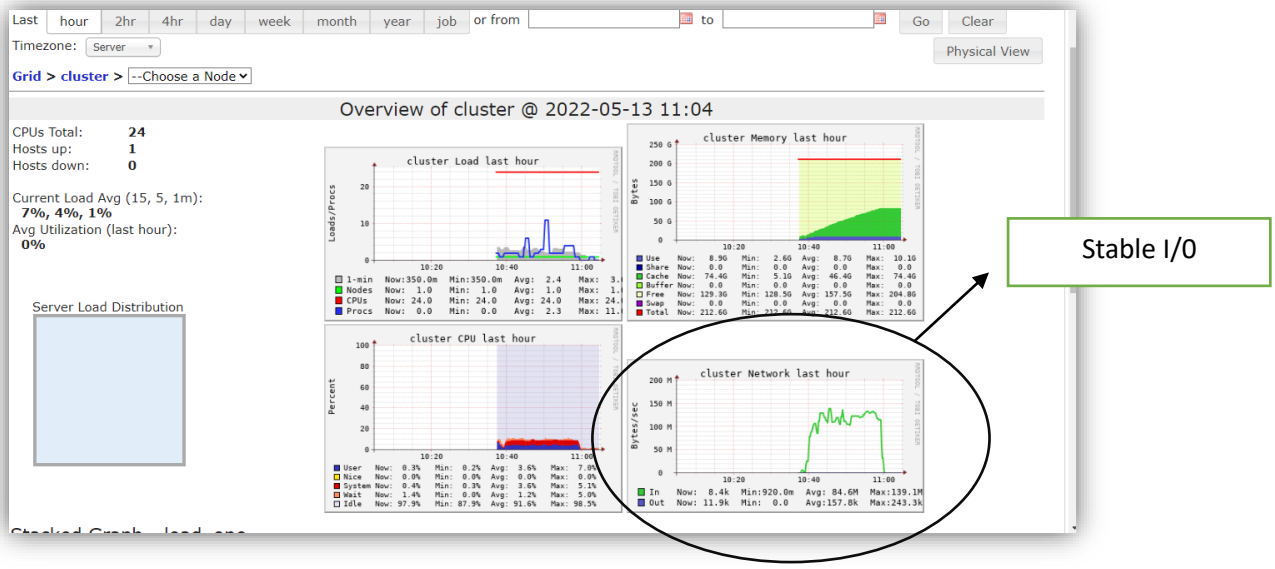

Figure 10: Training on CPU Machine with Optimized I/O

Summary

When building Deep Learning models, cloud compute and storage choices will make a significant difference in your compute costs and the overall time you need to spend waiting on your training iterations. Good design can generate significant savings. In this example, optimizing storage I/O increased your training speeds by nearly 6 times. The dollar savings were even more impressive. Using the default Databrick billing rates for NA region, the CPU machine was nearly twice as expensive as the GPU machine because it had higher RAM and CPU count. Choosing the right compute environment and optimizing the data layer accordingly would have reduced your cloud costs by nearly 22 times. DeepIQ makes it simple to generate these efficiency gains. In this example, DeepIQ workflows automated the conversion of data into an optimized Delta table format and DeepIQ ML pipeline automated the parallelization of storage I/O during the training process. With DeepIQ, your data scientists can focus on building world class machine learning models instead of worrying about the tedious task of optimizing cloud workloads.

For further information, please contact us at info@deepiq.com