Studying Storage System Backends for ML workloads

Benita Kathleen Britto, Hemalkumar Patel, Reetuparna Mukherjee, Group 13

1 Progress

Filesystems such as ext4 are commonly used for storing input datasets supplied to ML models. Although research has made significant contributions to improving ML training, it has not examined the impact of the storage format, the datastore, and the dataloader on ML workloads. The goal of our project is to analyze the preprocessing performance impact of various ML workloads, such as text, images, and graphs, by using three storage backends, RocksDB, TileDB, and TensorStore instead of using an ext4 filesystem, which is our baseline. It is this ML preprocessing pipeline that we investigate, which stores input data on these storage backends, retrieves data through a data loader, and transforms data before passing it to the training pipeline.

So far, we have implemented the storage and retrieval of the input data in these storage backends against different ML workloads. We mainly use Python to implement our project i.e. we use python interfaces to connect to the different storage backends and PyTorch to create a custom dataloader.

1.1 Infrastructure

We use a single-node instance of Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz on CloudLab.

1.2 Dataset

We used the Twitter sentiment analysis dataset for text [7], CIFAR-100 for images [1], and FB15K-237 for graphs [2]. To work on larger versions of these datasets, we concatenated the dataset with itself multiple times to understand the storage and retrieval implications as the size of the dataset increases.

1.3 Python support for storage backends

- RocksDB: We use the Rdict python library [3].
- TileDB: TileDB comes with the official Python library support [6].
- TensorStore: TensorStore comes with Python API support [4].

1.4 Storage Format

Text Dataset

- RocksDB: We store the data as key-value pairs wherein the key is set as the row index in the file and the value is the data stored at that row index in the dataset. Keys and values are stored as bytes.
- TileDB: TileDB provides the dimensions support for data. Since, there is only 1 dimension, i.e. datum index, we used that as a TileDB array index. The cell value of an array contains attributes of a tweet.
- TensorStore: TensorStore API provides an in-memory KVStore. We use the row index as the key and convert the attributes of the data tuple into bytes to store as the value.

Image Dataset

- RocksDB: We store the image sequence number as the key and the image tensor as the value (in bytes) after concatenating the image and the label and then, flattening it.
- TileDB: We repeat what we did for the Text dataset, i.e. choosing the datum index as the TileDB array index. However, in the case of an image, there are 2 attributes, "label" representing the image label and "image" representing the actual image. In each TileDB array cell, we stored "label" and "tuple of image values". A tuple is the only way to store many values in one attribute in a cell.
- TensorStore: We used the N5 dataset and file KVStore provided by TensorStore for images. We flattened the 32x32x3 image tensor and concatenated the label to form a 1x3073 tensor and stored it under the key corresponding to its datum index.

Graph Dataset

- RocksDB: We store the data as key-value pairs wherein the key is set as the row index in the file and the value is the data stored at that row index in the dataset (i.e. head, relation, and tail triplet).

- TileDB: We repeated what we did in the text dataset with the difference being the attributes. We used "head", "tail", and "relation" as an attribute for a cell.

- TensorStore: We store the data as key-value pairs in the underlying KVstore, similar to the approach for RocksDB. The key is the datum index and the value is the triplet converted to bytes. We also plan to experiment with N5 and Zarr drivers to see if we can store strings (converted to bytes). As per our understanding from the documentation [5], the drivers seem to support only number formats.

1.5 Addendum

We also explore the impact of storing intermediate data on these storage backends. The embeddings generated from the input data are considered intermediate data. We add this aspect of an addendum to our initial project proposal.

Storage format for graph embeddings

- RocksDB: We store a generated head embedding key as the key and the embedding in bytes as the value. We follow a similar process for relations and tails.

- TileDB: For embeddings, we create 2 TileDB arrays, one for vertex and one for relation with the D length cell for each array.

- TensorStore: We store the embeddings as tensors - one for vertex and one for relation, similar to that in TileDB.

1.6 Dataloader

We use iterable-style and map-style dataloaders using PyTorch's torch.utils.data module. An iterable-style dataloader prefetches data while a map-style dataloader fetches data a single record at a time.

1.7 Observations

- While the ML algorithm we are using might differ on a case-by-case basis, the pattern remains almost the same. For example, for Image, we usually read data and transform the image before passing it to the model. For text, we do 2 iterations. The first step consists of embedding generation. We either generate embeddings with a custom algorithm or use the existing word2vec, which in itself is a neural network, and create embeddings. The second step involves reading the data, accessing the embeddings, and then passing it to the model. The process for graphs is similar to the text workload.

- The goal of the transformation step of the preprocessing pipeline was to pushdown the transformation to the storage backend. However, no storage system provides transformations as pushdown operations other than filters. This filtering capability is only offered by TileDB. Pushing down transformations is crucial for workloads like images which involve rotation, padding, and other operations. As these operations cannot be done by the DB itself (i.e. no pushdown), they are performed by the CPU.

- The datasets we consider have different datatypes and so, they can provide a comparative study of persisting different datatypes.

- While the Text and Image dataset we used involves sequential read operation (loading the data from the storage backend), the Graph dataset involves a mix of sequential read, random read, and random updates during embedding generation. This is not a universal truth, but it is true for our dataset and the ML algorithm we are using for each of them.

2 Challenges

In the final step of the preprocessing pipeline, data is transformed on retrieval from the dataloader before being passed on to the machine learning model. For example, in the Twitter sentiment analysis dataset, stop word removal is part of the data transformation step. It was our intention to push down such transformation to the storage backend. These transformations, however, aren't supported by the storage backends we're looking at. Data filtering, offered by TileDB, is the only transformation that can be applied and so, is not useful to the ML workloads we are investigating - text, image, and graph. Therefore, we do not include this component in comparing storage backends with filesystems. Rather, we will explore how embeddings are stored and retrieved across different access patterns. We thus extend our project by examining the impact of storing intermediate data in these storage backends on performance.

Another challenge we faced was the lack of documentation for TensorStore. We could not find any paper to understand the design which could help us explain our observations with different datasets. The official tutorial does not provide enough explanation or examples to clarify how or when different datasets (N5, Zarr, etc) and configurations should be used. Also, TensorStore tutorial suggests that we use the N5 or

Janelia FlyEM Hemibrain as the dataset, both of which support only number formats. Handling string (text and graph) data, therefore, is not straightforward, and cannot leverage the benefits that TensorStore provides. We use the underlying KVStore directly to store the strings converted to bytes. This limits us to reading/writing only one key-value pair at once, unlike bulk fetch in the case of numerical tensors (used for image data).

3 Timeline

We plan to measure the performance implications of these storage backends on input and intermediate data for the different ML workloads. We will be measuring the read time, write time, memory footprint, and CPU usage across our tests and tuning parameters like the number of workers, the batch size, and the prefetch size (iterable style dataloader). For intermediate data, we will be documenting the performance against different access patterns like read-only, write-only, read-skew, write-skew, etc on embeddings that are generated through a well-known distribution or generating embeddings from the existing datasets that we have considered so far.

Our timeline for the rest of the semester will be:

- Now - 11/26: Input data measurement
- 11/27 - 12/3: Intermediate data measurements
- 12/4 - 12/10: Measurement wrap-up, analysis
- 12/11 - 12/13: Poster presentation preparation
- 12/14 - 12/20: Report

References

[1] Cifar-100 dataset https://www.kaggle.com/datasets/fedesoriano/cifar100.

[2] Fb15k-237 dataset https://paperswithcode.com/dataset/fb15k-237.

[3] Rocksdb python library https://github.com/congyuwang/rocksdict.

[4] Tensorstore python library https://google.github.io/tensorstore/python/tutorial.html.

[5] Tensorstore zarr https://google.github.io/tensorstore/driver/zarr/index.htmlmapping-to-tensorstore-schema.

[6] Tiledb python library https://docs.tiledb.com/main/how-to/installation/quick-install.

[7] Twitter dataset https://www.kaggle.com/datasets/kazanova/sentiment140.