Studying Storage Systems for ML Workloads

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

1 Introduction and Background

Our project aims to compare the preprocessing performance of various machine learning workloads, such as text, images, and graphs, using three storage backends (RocksDB, TileDB, and TensorStore) to using an ext4 filesystem as a baseline. The ML preprocessing pipeline stores input data on these storage backends, retrieves it through a dataloader, and transforms it before passing it to the training pipeline. Ext4 filesystems are commonly used for storing input datasets for ML models, but research has not thoroughly examined the impact of the storage format, data store, and dataloader on ML workloads.

2 Related Work

Existing literature explores various arenas for improving performance of ML processing systems and frameworks ranging across caching capabilities and IO optimizations.
Caching: Quiver [9] achieves cache reuse across multiple jobs securely. As a result, even when only a small part of data fits in cache, Quiver improves performance by using substitutable hits and co-operative miss handling to co-ordinate I/O access across multiple jobs. It uses a benefit-aware strategy in its placement thereby using the cache frugally and prioritizing jobs that benefit the most. Distributed caching in the cluster context has been explored more broadly in the analytics community. To extract the best performance from a query, Pac-Man [7] explored co-ordinated caching of data across different workers of an analytics job. Quartet [8] explores intelligent scheduling of big data jobs to maximize cross-job sharing of cached data.
I/O optimization: DeepIO [14] primarily features an I/O pipeline that utilizes several novel optimizations (RDMA -assisted in-situ shuffling, input pipelining, and entropy-aware opportunistic ordering) and additionally provides a portable storage interface to support efficient I/O on any underlying storage system. Crail [13] is a fast multi-tiered distributed storage system for high-performance network and storage hardware to deliver user-level I/O.

Data Stall [11] analyzes how data stalls affect the data pipeline and DNN training. Their analysis suggests that data stalls take away the benefits of faster GPUs, even on ML optimized servers like the DGX-2. The modern ML frameworks like PyTorch and TensorFlow utilizes the state-of-the-art dataloaders like Dali, which reduce prep stalls using GPU-accelerated data pre-processing. Even then they are inefficient in their use of memory and CPU resources.

Dataloaders: Ofeidis et al. [12] go over the existing libraries and the dataloaders they offer and compare their performance against three datasets CIFAR-10 (can fit in memory), CoCo, RANDOM (created by the authors by augmenting CIFAR-10). Using frameworks like PyTorch, Torchdata, Hub, FFCV, WebDatasets, Squirrel, and Deep Lake, they show that the dataloaders provided by a certain framework are not optimized for all workloads when run on limited resources. Metrics collated over the training data, like the number of processed data points (images) per second, total running time, and data loader initialization, and further tuning the dataloader on hyperparameters like batch size, sampling approach, collating/padding to link batches, prefetching, number of workers, number of GPUs show the performance of dataloaders varies based on the workload. They also test when data is stored remotely and gather that remote storage does not hamper performance by a lot. Storing data in a remote storage environment is beneficial as the data setup becomes more streamlined. LBANN, on the other hand, uses node-local storage devices to store datasets, but not all clusters feature the expensive fast-speed node-local storage devices [10].

To the best of our knowledge, the related work focuses on improving the data pipelines and stalls, and does not explore the impact of the underlying storage systems. Therefore, we plan to understand how the underlying storage layer can impact the performance of ML tasks under different types of workloads.

3 Approach

3.1 Infrastructure

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

3.2 Dataset

We used the Twitter sentiment analysis dataset for text [6], 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.

3.3 Python support for storage backends

- RocksDB: We use the Rdict python library [3].
- TileDB: TileDB comes with the official Python library support [5].
- TensorStore: TensorStore comes with Python API support [4].

3.4 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: As per our understanding from the documentation, the TensorStore drivers seem to support only number formats, which is why we resort to using the underlying KVstore, similar to the approach for text workload. The key is the datum index and the value is the triplet converted to bytes.

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 embeddings as tensors - one for vertex and one for relation, similar to that in TileDB.

3.5 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 embeddings

- RocksDB: We store a generated head embedding key as the key and the embedding in bytes as the value.
- TileDB: For embeddings, we create TileDB arrays.
- TensorStore: We store the embeddings as tensors.

3.6 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.

3.7 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 specific to our dataset and the ML algorithm we are using for each of them.

4 Experimental Setup

We experiment with the following parameters for the input data:

- Map style dataloader: Batch size, Rows per key (Store a batch of input rows per key in RocksDB only), Number of workers
- Iterable style dataloader: Batch size, Prefetch size, Number of workers

We experiment with the following parameters for intermediate data: Dataset types (Zipf distribution for text, Power Law distribution for graph), Embedding size, Batch size

Acronymns used throughout our experiments are in section 9.1.

5 Results

5.1 Write Overhead

Setup: With RocksDB, we experiment with storing multiple rows per key to reduce the write cost. We use multiple workers to write data to TileDB and certain configurations of TensorStore. We do not use multiple writers for RocksDB as write locks were causing write lock issues. The TensorStore N5 driver uses multiple workers internally. We use the async capability of TensorStore to speed up writes.
Observations: An added overhead with using data stores is the time taken to write the data to each of the data stores. In this section, we analyze the write time for each of the workloads as shown in Figure 1.

For the text workload, RocksDB with a single key-value configuration is slow as expected. It improves a lot

Figure 1: Cost of converting dataset to DB format

with multiple rows under the same key (denoted by R) as the number of calls to disk reduces. TileDB performs the best as it uses multiple workers to write data. Performance of TS with single key and single value is very close to RD's best configuration.

For the image workload, TensorStore outperforms all the other data stores as expected since it is built for numeric data types and uses multiple workers to parallelize the process internally. We use the N5 driver provided by the TensorStore API and tune the dimensions, chunk size and compression configuration for the best performance. A significant takeaway though is that RocksDB is not too far from the optimal performance. Surprisingly, TileDB does not perform better than RocksDB (when R != 1) in spite of using multiple workers.

For the graph workload, the results are pretty similar across all the datastores, except for RD R = 1, due to the number of disk calls for a small sized writes. We also note that the dataset is quite small and fits in memory.

5.2 Text Workload - Map-style Dataloader

Setup: We experimented with the PyTorch map-style dataloader with different numbers of workers and varied batch sizes. For RocksDB, we store multiple rows per key and tune this parameter. We experimented with storing multiple rows per key only for RocksDB as storing one row per key for this datastore had poor performance when compared to the other datastores. TensorStore supports in-memory storage for text data.
Observations: Increasing the number of workers definitely improves performance but only to a certain point after which we observe diminishing returns on further adding workers. For text workload, 8 workers is where we see the most optimal results. Thus, the parallelism saturates after 8 workers for this workload. RocksDB performs the best over all configurations, even beating TensorStore's in-memory access time as shown in Figure 2! In Figure 3, we vary the number of rows per key stored in RocksDB. We get the best load time with rows

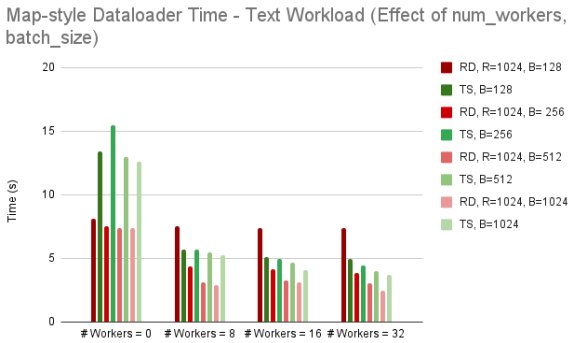

Figure 2: Map-style Dataloader Time - Text Workload

per key of 512 and use this result in the previous graph to compare its performance with TileDB.

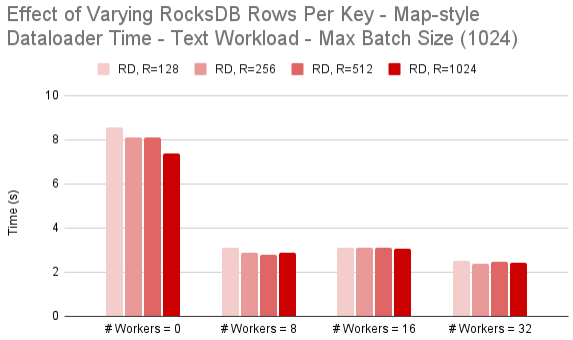

Figure 3: Effect of Varying RocksDB Rows Per Key - Text workload

5.3 Text Workload - Iterable-style Dataloader

Setup: We repeat the same experiment with the iterable-style dataloader as well. With the iterable style dataloader, we have another parameter to take into consideration - the prefetch size. We first fix the batch size and vary the prefetch size, and then repeat the same for different batch sizes. For RocksDB, we do not experiment with storing multiple rows per key as we use prefetching across all datastores. We do not experiment with TensorStore as it does not support iterable-style access with text data.

Observations: The above graphs (Figure 4) show that irrespective of the batch size, for the chosen text dataset, prefetch size of 256 always performs the best. Similar to the Map-style dataloader, we see optimal results with workers 8-16, which confirms the saturation of the parallelism for this workload. TileDB performs much better

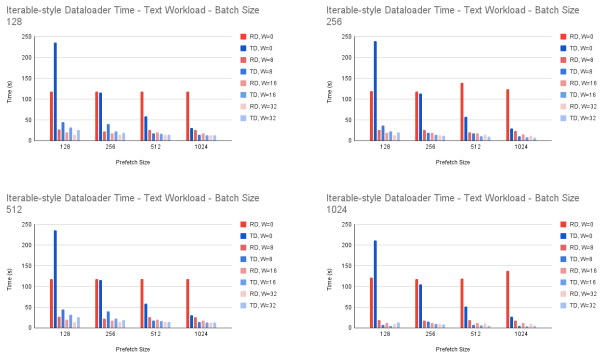

Figure 4: Effect of varying batch size in iterable dataloader for text workload

than RocksDB when using iterable-style over map-style due its efficient prefetch capability.

We also see that RocksDB with multiple rows per key (map-style dataloader) and prefetch size perform differently. We use the multi_get API to fetch values of multiple keys (iterable-style dataloader). This could mean that multi_get results in many more I/O calls than storing multiple rows per key. We see the same trend across all workloads (text, image, graph) when using iterable-style for RocksDB.

5.4 Image Workload - Map-style Dataloader

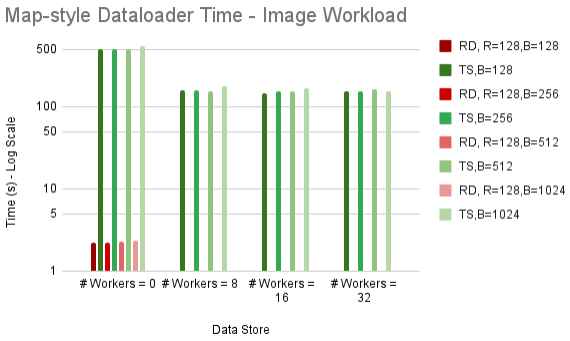

Figure 5: Map-style Dataloader Time - Image Workload

Setup: We do not show the results for TileDB as it was extremely slow in comparison to the other data stores. We repeat the same setup as text map-style except for TensorStore, which uses disk storage as this workload deals with numeric data.

Observations: TileDB (not shown in the graph) was the slowest, while RocksDB performs the best (which is seen as gaps in the graph) as shown in Figure 5. Thus, show-

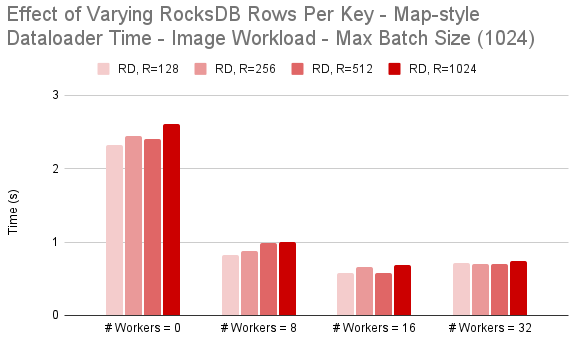

Figure 6: Effect of Varying RocksDB Rows Per Key - Image workload

ing a trend that RocksDB map-style is almost always better than TileDB and TensorStore.

We also show the effect of varying rows per key in RocksDB (Figure 6). The results are consistent with that of the text workload. Increasing the number of workers beyond a number, 16 in this case, does not yield any better results.

5.5 Image Workload - Iterable-style Dataloader

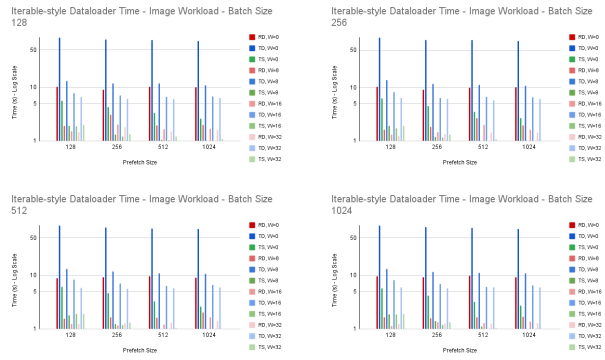

Figure 7: Effect of varying batch size in iterable style dataloader for image workload

Setup: Same as text iterable-style except for TensorStore which supports iterable access.

Observations: TensorStore always outperforms the other stores (Figure 7). TensorStore is built for multi-dimensional numeric arrays and batch reads are significantly optimized compared to other stores.

5.6 Graph Workload - Map-style Dataloader

Setup: Same as text map-style.

Observations: TensorStore performs the best over all datastores (Figure 8), which is unlike the text workload results. This can be attributed to TensorStore using in-memory storage and the graph workload being 1/10 of the text workload. We also see similar trends with increasing the number of rows per key for RocksDB as we saw previously with the text and image workload (Figure 9).

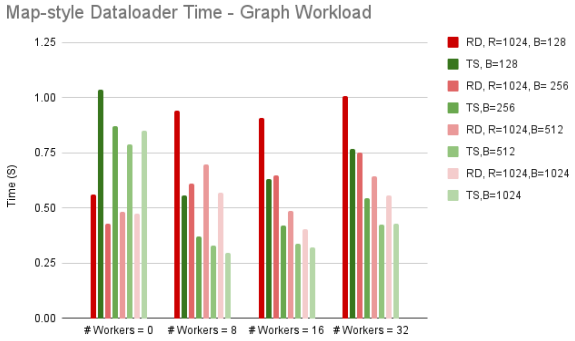

Figure 8: Map-style Dataloader Time - Graph Workload

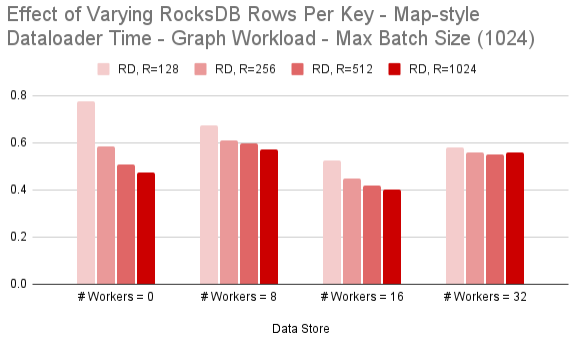

Figure 9: Effect of Varying RocksDB Rows Per Key - Graph Workload

5.7 Graph Workload - Iterable-style Dataloader

Setup: Same as text iterable-style.

Observations: TileDB performs better than RocksDB only with a number of workers over 8 (Figure 10). Yet again, the batch size not affect the performance.

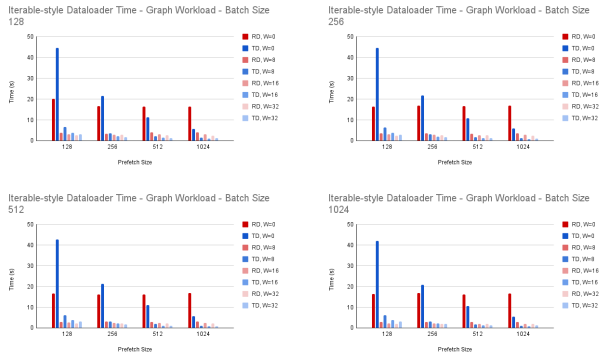

Figure 10: Effect of varying batch size in iterable style dataloader for graph workload

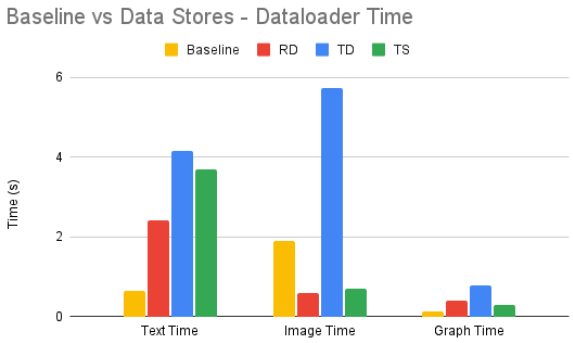

Figure 11: Baseline vs Data Stores - Dataloader Time

5.8 Summary of Dataloader Results

Finally, we compare the best results obtained from the above-mentioned experiments to the baseline (ext4 filestystem) as shown in Figure 11. We see that data stores - RocksDB and TensorStore - outperform the baseline only for the image workload. The image dataset size is much larger than the text and graph datasets.

The best configuration for the data stores is mentioned in Table 1.

5.9 Input Scalability

Setup: Since our original dataset FB15k-237 contained only 272K triplets (21 MB). We augmented it with duplications, varying the dataset size from 2x to 50x and appending it repeatedly. Similarly, we augmented the text dataset (originally 228 MB) up to 10x. This served our purpose for increasing the size of the dataset up to 1GB with no change to the code, and still being truthful about our results, since we are not training our models at all. This would expose any underlying caching benefits.
We measured 2 aspects:

Workload	RD	TD	TS
Text	R=1024, W=32, B=1024, T=M	W=16, B=1024, PF=1024, T=I	W=32, B=1024, T=M
Image	R=128, W=16, B=1024, T=M	W=32, B=512, PF=512, T=I	W=32, B=512, PF=512, T=I
Graph	R=1024, W=16, B=1024, T=M	W=16, B=512, PF=1024, T=I	W=8, B=1024, T=M

Table 1: Configurations that gave the best result for each db in each workload type

- Write overhead - time taken to store the new dataset to DB
- Dataloader time - time taken to load the entire dataset from DB using dataloader

The results are shown in Figure 12.

Observations: RocksDB uses LSM trees which is really fast for write-only workloads. TensorStore uses machine's memory, but somehow is not as performant as RocksDB. RocksDB and TileDB performs similarly and is better than TensorStore for the graph workload.

The default cache store size for RocksDB is 50MB and for TileDB is 10 MB. We did not vary these parameters to see their effect to honor the brevity of this experiment. TensorStore does not provide any direct mechanism to store strings/text data, so we had to rely on its in-memory KV store. This could be the reason the access times are not as optimized as expected for the text and graph workloads (which contain text data). For the image workload, TensorStore's N5 driver performs the needed optimizations under the hood. We cannot confirm this owing to lack of design documentation on TensorStore.

Our 1x text file size (text workload) is 228 MB, which is already greater TileDB's cache and takes extremely long, and so we do not show the results for it. At 2x file size, we reach the limit of RocksDB cache size. TensorStore is irrelevant in this conversation, as it uses the machine's memory which is too large anyway.

For the graph, TileDB and TensorStore scale really well giving similar performance on different dataset sizes. Even with input size over 1GB (50x), this trend continues. For baseline and RocksDB, read time keeps increasing steadily with an increase in input-size, and RocksDB performs worse than baseline. The write-overhead, on the other head, increases with increasing input-size for all the DBs, with TileDB being the slowest.

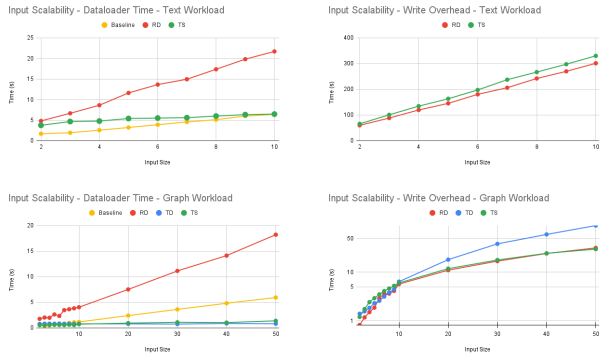

Figure 12: Read and Write time scalability. Left column shows dataloader time for different dataset size for different dbs. Right column shows cost to transform dataset to db format. 2x means twice the size of dataset used in that workload.

5.10 Embedding

Setup: Text dataset follows zipf distribution and graph follows power law distribution. So we generated a dataset that follows those distributions. We call it dataset_size. Next, we varied the encoding/embedding size for each dataset. We call it embedding_size. This embedding size determines the length of the encoding per each unique input.

Our setup feeds PyTorch iterable-style dataset to PyTorch Dataloader. This dataloader iteratively reads the batch of data from the file. It then fetches the embedding from the DB and generates new embedding for each. Generating new embeddings is synonymous with passing the batch to the model and compute new embedding. Then, we updates the DB with those new embeddings.

Our experiment focuses on the impact of the 3 parameters essential for this experiment. We fixed 2 parameters, and varied another to see the impact of it.

Observations:

Figure 13: Effect of dataset size keeping batch size and embedding size constant. left=text, right=graph

As shown in Figure 13, the total time taken by all data stores increases linearly with increasing dataset size. This is because we do not use any workers in dataloader so fetch time increases linearly. Moreover, DBs don't show any significant difference in reads/writes.

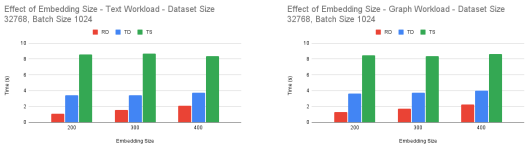

Figure 14: Effect of embedding size keeping dataset size and batch size constant. left=text, right=graph

Changing embedding size means that dbs have to store more information pertaining to each key. As shown in Figure 14, TensorStore and TileDB does not show any change if we increase the embedding size. However, RocksDB shows increase in time as embedding size increases. We contribute this to the underlying storage mechanism of TensorStore and TileDB, as they are designed specifically for ML workloads where embeddings of long length are not uncommon, on the other hand RocksDB is more of a generic KV-store and not targeted for ML workloads.

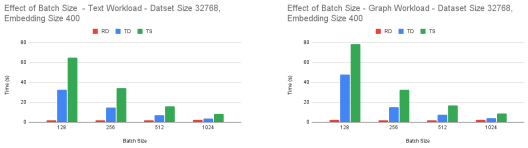

Figure 15: Effect of batch size keeping dataset size and embedding size constant. left=text, right=graph

Changing batch size has 2 effects. First, we fetch more data during batch read from dataloader. Second, the reads/writes to the DBs contains more keys for which we want to read/write embedding. As shown in Figure 15, We can clearly see that it has no impact on RocksDB as it follows "everything is a key/value" and keys have no neighborhood relationship. This results in the same total time as fetching 2 values using multi-get twice or 4 values using multi-get once. However, TileDB and TensorStore shows interesting results. Since they are designed for ML workloads they seem to honor the neighborhood relationship between keys. Because of which as we fetch more data in a single operation using increased batch-size, their total time also reduces by half each time. At batch_size=1024, for TileDB it is almost similar to RocksDB, and for TensorStore it is slightly higher than TileDB.

5.11 CPU and Memory Overhead

Setup: We used the best configurations across all datastores to capture the CPU and memory overhead.

Observations:

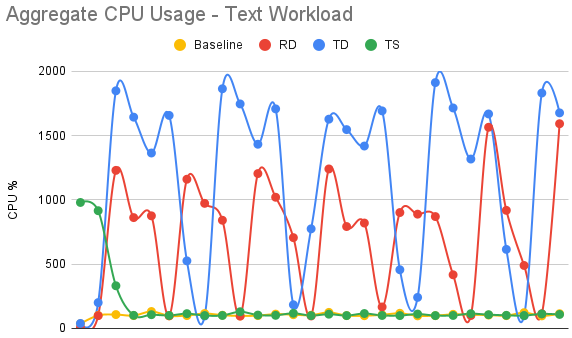

Figure 16: Aggregate CPU Usage for Text Workload

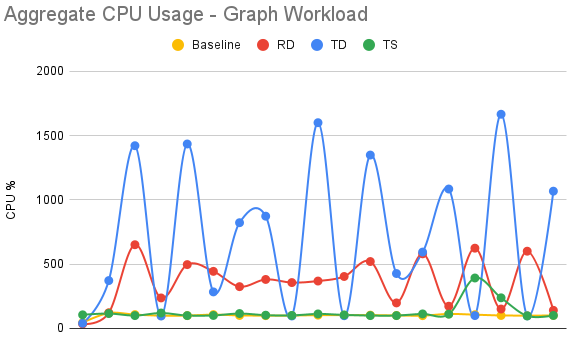

Figure 18: Aggregate CPU Usage for Graph Workload

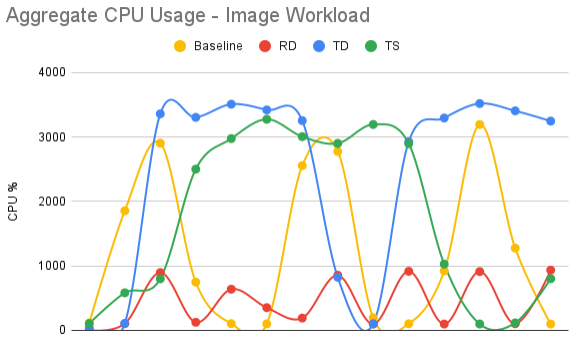

Figure 17: Aggregate CPU Usage for Image Workload

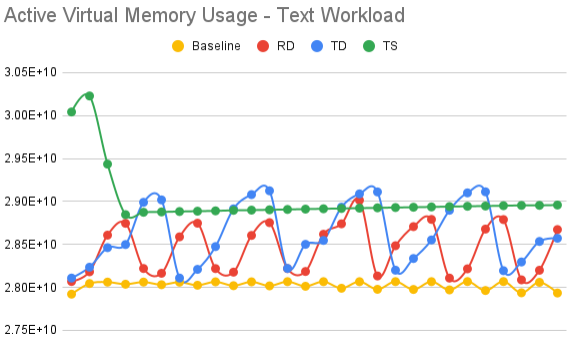

Figure 19: Memory Usage for Text Workload

Figures 16-21 show the CPU and memory usage of the different DBs.

- For text and graph workloads, baseline and TensorStore have less than 1000% aggregate CPU usage, while the usage for baseline and TileDB spike in intervals. This can be attributed to multiple workers leading to higher degree of parallelization in TileDB. TS starts with a higher usage but then drops to same as baseline.
- For image, TensorStore and TileDB reach over 3000% aggregate CPU.
- TensorStore has an almost constant virtual memory usage for each workload.

6 Future work

As part of the future work, we can experiment with storing multiple rows per key/index in datastores TileDB and TensorStore for the map-style dataloader configuration. We can also experiment with storing multiple rows per key for all datastores using iterable-style dataloaders. We can also explore other workload datasets for text, image and graph. Another interesting thing would be to see how the performance varies with update patterns - sparse vs dense updates, or locality of updates, etc.

7 Contributions

The contributions of our team members are as follows:

- Benita Kathleen Britto: RocksDB, experiments related to dataloader time
- Hemalkumar Patel: TileDB, experiments related to embeddings
- Reetuparna Mukherjee: TensorStore, experiments related to scalability

8 Conclusion

We have explored the impact of using RocksDB, TileDB and TensorStore on dataloader times and fetching and storing embeddings. We tuned our experiments using multiple parameters like batch size, prefetch size, number of workers, rows per key, map-style dataloader and iterable-style dataloader. We observe that it is possible for some of these datastores to outperform the dat-

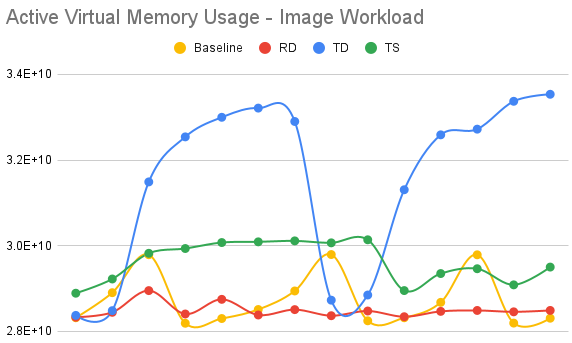

Figure 20: Memory Usage for Image Workload

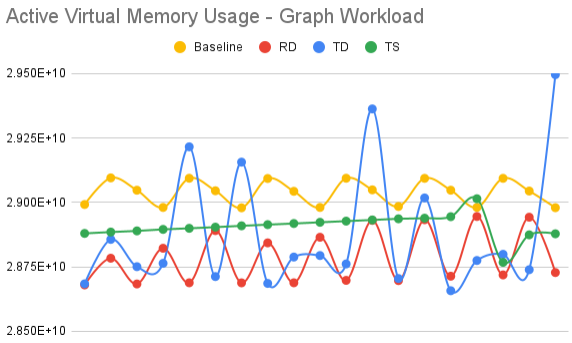

Figure 21: Memory Usage for Graph Workload

aloader times of a baseline filesystem. However, this depends on the type of workload and the size of the dataset. While TensorStore is very good for very large numerical datasets, RocksDB can be modified to store multiple values under a key to yield the best performance for text data. This also comes with an additional one-time cost of storing the data in the datastore.

References

[1] Cifar-100 dataset.

[2] Fb15k-237 dataset.

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

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

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

[6] Twitter dataset.

[7] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. {PACMan}: Coordinated memory caching for parallel jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, 2012.

[8] F. Deslauriers, P. McCormick, G. Amvrosiadis, A. Goel, and A. D. Brown. Quartet: Harmonizing task scheduling and caching for cluster computing. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[9] A. V. Kumar and M. Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, 2020.

[10] N. Maruyama. Scalable distributed training of large neural networks with lbann. 2019.

[11] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020.

[12] I. Ofeidis, D. Kiedanski, and L. Tassiulas. An overview of the data-loader landscape: Comparative performance analysis. *arXiv preprint arXiv:2209.13705*, 2022.

[13] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[14] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. Multi-client deepio for large-scale deep learning on hpc systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2018)*, 2018.

9 Appendix

9.1 Acronymns

Across the experiment graphs shown below, we use the following acronyms:

- B: Batch size
- I: Iterable-style dataloader
- M: Map-style dataloader
- R: Rows per key
- RD: RocksDB
- T: Dataloader type

- TD: TileDB
- TS: TensorStore
- W: Number of workers

9.2 Range of Dataloader Times

Below is the range of dataloader times we see for the different workloads by collecting the times across all the experiments we ran. A peak denotes that certain parameters were not well suited for the datastore. We observe a larger amount of readings for RocksDB because we experimented with an additional parameter for this datastore (rows per key).

Figure 22: Range of dataloader times for all Datastores across all workloads

9.3 Code Repository

https://github.com/benitakbritto/CS744-Project