MorpheusFlow: a case study of learning over joins with TensorFlow

Side Li Arun Kumar
University of California, San Diego
La Jolla, California
{s7li,arunkk}@eng.ucsd.edu

ABSTRACT

In real world, many datasets are stored as multiple tables in a normalized fashion in relational databases management system (RDBMS). Most machine learning (ML) toolkits expect the training dataset to be a single table with all the features. This forces data scientists to often perform joins to materialize a single table before ML that concatenates features from all base tables, creating lots of redundancy on disks. In MorpheusFlow, we address this space deficiency problem by studying lazy join, a means of dynamically joining tables closer to ML modeling. We trade off lazy join versus eager join. We adopt common practices that can improve lazy join's performance. We propose a Dataset library built with TensorFlow that dynamically joins multiple tables into mini batches, which can directly get fed into an SGD-based (Stochastic Gradient Descent) model. Our results show that lazy joins are only marginally slower than eager joins in simple linear models and shallow neural networks while saving much more spaces.

KEYWORDS

Machine Learning, Join, TensorFlow, Data Pipeline

1 INTRODUCTION

Big data has emerged into many fields presenting massive sources including both structured (relational) and unstructured (images, music, metadata, etc.) data. Along with advancements in computations, machine learning (ML) has also taken off recently. Many companies are racing to exploit their massive data using ML techniques. In many cases, their data are stored as multiple tables in a normalized fashion in relational databases management system (RDBMS). However, most ML toolkits only expect a simple table with all necessary features as training input. For example, TensorFlow [1] disregards normalization data because it only consumes one single table either from memory or disk. TensorFlow is one of the most widely used ML frameworks that uses dataflow graphs in computation. The design of ignoring normalization forces data scientists to perform joins to materialize a single table before ML modeling, ruining benefits of data normalization. The dataset might blow up in sizes due to the introduction of redundant entries from joined tables. Such redundancy also brings the burden on both ML pipeline and modeling cost. This necessitates the need of studying when and how should we join multiple tables before modeling such that less scalability, performance will be sacrificed.

In recent work, Kumar el al. [4] studied the effect of joins over normalized data on linear models. Their work indicates whether to join before or at ML modeling is a decision depending on available resources. It is not necessarily slower to do lazy joins at runtime versus eager joins. Eager joins means materializing a single table before ML modeling while lazy joins means dynamically building joined data in ML data pipeline. Nonetheless, their study applies only to a setting of RDBMS. Nowadays, most ML scientist and engineers work with ML frameworks in a high-level client language. Python is one of the most popular languages in the current trend of ML development. As such, we extend the idea of learning over joins to a new environment in Python and TensorFlow. In summary, we explore trade-offs of lazy joins and eager joins. We also experiment with techniques that would improve the performance of lazy joins. We engineer a Dataset library for TensorFlow that could easily adopt normalized data inputs into ML pipeline. Finally, we present heuristics of deciding whether to use lazy join from modeling's perspective.

Outline. The rest of the paper is organized as follows: In Section 2, we review the background of this work. In Section 3, we discuss design principles of Dataset library. In Section 4, we illustrate the implementation of Dataset library. In Section 5, we validate our implementation with synthetic and real data sets.

2 BACKGROUND

2.1 Data Systems

Big data systems such as Spark[6] and Hadoop[5] have been widely integrated with ML frameworks. These systems serve as underlying data lakes that host massive raw data. It is up to data engineers or software engineers to query the systems to materialize parts of training data into files. Data scientists will then make use of data files to train the model. In a nutshell, querying the data lake is a stage before putting data into ML pipeline. As such, integration with these data systems is not the concern in this project. We assume normalized data have already been loaded onto disks or memory.

2.2 Stochastic Gradient Descent

ML models usually have various parameters and a corresponding objective function. The goal of ML modeling is to optimize the objective function over parameters and training data. Stochastic Gradient Descent (SGD) has recently been widely adopted as the optimization method, because of its ability to optimize very large datasets. Instead of updating coefficients per epoch, SGD updates coefficients per training batch. In other words, SGD only takes a mini batch at each epoch. We study whether joining at runtime during mini batching will give us comparable runtime performance as materializing joins before modeling.

Recently, lots of variations of SGD such as Adam[3] have been studied to provide faster and stabler convergence. The study of SGD optimizer is orthogonal to MorpheusFlow, as we focus on optimization at data sourcing stage. Given different optimizers still

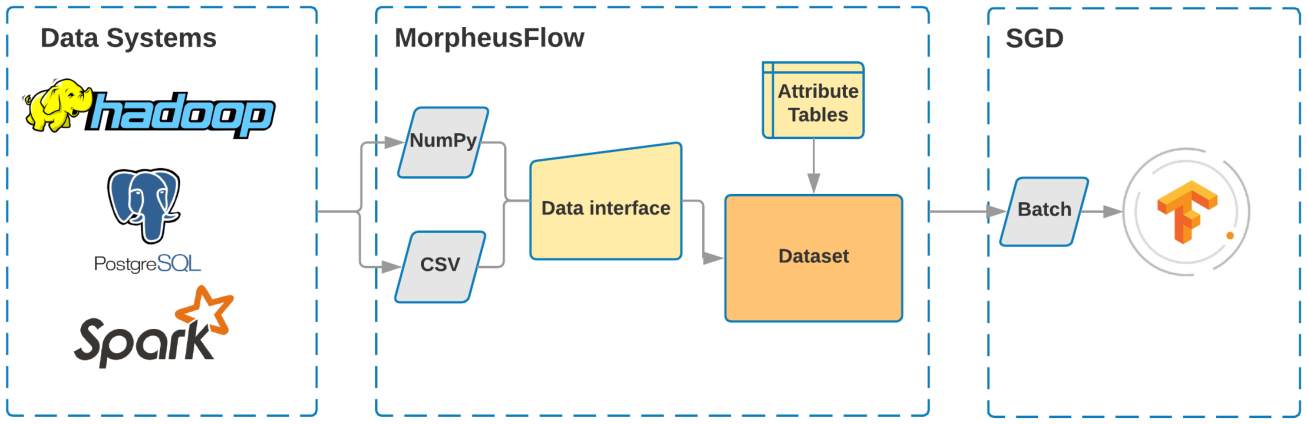

Figure 1: MorpheusFlow Overview

consume mini batches, we choose to experiment with SGD in this project for the sake of simplicity.

2.3 Lazy Joins vs Eager Joins

The intention of comparing two types of joins comes down to deficiency of available resources and ease of development. If datasets can comfortably fit into memory, eager joins should usually be the top choice. However, nowadays scientists deal with massive materialized datasets that can hardly fit in memory, necessitating strategies to swap data into memory when needed. Lazy joins suffice this criterion by consuming only a small chunk of data from a file and then dynamically performing a join on the batch. Then, it boils down to a two-fold reason for using lazy joins: first of all, less space is used. A single materialized table might take up too much space on disks because of redundancy brought by denormalizing relational datasets. Second, practically speaking lazy joins decentralize responsibility of joining tables from data systems to ML training node. It avoids the situation of people under one or multiple organizations waiting for data systems to produce the joined table.

3 DESIGN PRINCIPLES

A dynamic join strategy like lazy joins will unavoidably cause additional runtime overhead. Therefore, in this project, we mean to reduce the overhead as much as possible while still preserving the benefits of space efficiency. We seek for optimization strategies in joins and apply them in MorpheusFlow. We create simple abstractions that are easy to use with existing normalized data files.

3.1 Multiprocessors

Multicore design in CPUs becomes prominent as we are getting closer to the limit of the single processor. For instance, lots of commodity machines are equipped with many cores even up to hundreds of cores. Usually, in ML library, only single core might be designated to read data and so join tables, which is a waste of hardware resources. As such, we explore triggering multiple cores to process joins to boost up lazy joins.

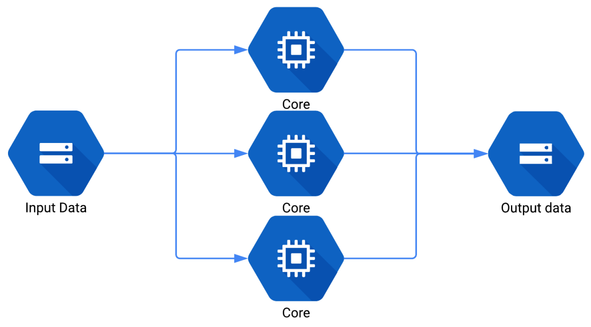

Figure 2: Multicore usage in lazy join

3.2 Batch

The technical reason for lazy joins bringing overhead is that it causes more system overhead per data entry. Each time the system processes an entry, we need to look up attribute tables to find corresponding rows to join. Meanwhile, lots of context switches will have passed. As a result, the total throughput of data pipeline is low. Usually, in the dataset, there might be all kinds of locality that favor usage of caches. We, therefore, experiment with batch joins to keep a high rate of system usage.

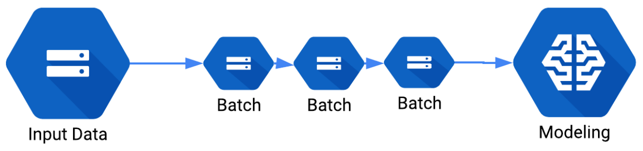

Figure 3: Batch usage in lazy join

3.3 Abstraction

As an abstraction that connects normalized data files to batched input, MorpheusFlow should also ensure ease of usability and extensibility. Foremost, it should support bare NumPy array and SciPy sparse matrices that data scientists have been using. Also, reading from CSV file should also be supported since it is an almost standard format used in sharing data in open source community. Besides, the abstraction should also support easy integration to other file types such that users only need to implement few lines of codes.

4 IMPLEMENTATION

We implement MorpheusFlow on top of Dataset, a recently published library in Tensorfow. The overall design is presented in figure 1. The Dataset library serves as a logical entrance for ML pipeline to consume a nested collection of data. Our abstraction as a high-level wrapper of the native Dataset also gets easy access to native functions like shuffling, repeating and parallel processing. In addition, we implement a *get_next()* function to return a designated mini batch at a time. The key contribution in our implementation is that we dynamically perform lazy joins over a mini batch of entity table, and a set of attribute tables. We keep all attribute tables in memory, or in disks if memory is full while reading relatively large entity table a mini batch per join. Our abstraction can easily consume CSV, NumPy, TFRecord files, and any other popular matrix-like data formats. In a nutshell, our Data API takes in relational datasets (entity table and attribute table) and offers a series of mini batches as output.

5 EVALUATION

To measure the performance of MorpheusFlow, we conduct experiments on synthetic datasets and real-world datasets. We aim to study how much overhead lazy joins can impose on ML training and how MorpheusFlow can alleviate the problem.

5.1 Experiment setup

All experiments were run on CloudLab [2]. All experiments were conducted on a machine with 14 Intel Xeon E5-2683 2.0GHz cores, 224GB RAM and 3TB disk with Ubuntu 16.04 LTS as the OS. Our code is implemented in Python 2.6, and we use NumPy 1.13, SciPy 1.1, TensorFlow 1.5.0 as dependency libraries.

5.1.1 Datasets. For synthetic datasets, we have ns=100000, ds=50, nr=500 and dr=200. As for real-world datasets, we adopt some used in the Morpheus project including Expedia, Movie, Yelp, and Flights. Note that all real-world datasets are very sparse because all categorical data are one-hot encoded leaving lots of zeros.

5.1.2 Algorithms. We use basic logistic regression and naive neural network (2 hidden layers) in the experiment. These simple models are easy to track and therefore helps us study the interplay of I/O cost and computational cost. Further, our results should also be able to shed some lights on complex models given the patterns studied.

5.2 Results on Synthetic Data

We first examine performance difference in lazy join and eager join when we make more use of hardware resources - processors. Regarding setups, we use logistic regression (LR) and the basic SGD optimizer provided in Tensorflow. We study the average runtime cost per epoch on, for we are interested in how lazy joins would affect end-to-end ML training. The result is illustrated in figure 4. Overall, we see that lazy joins are generally slower than eager joins, most likely due to the overhead on dynamic joining. Concerning multi-core performance, we notice that two processes can substantially reduce the average runtime cost. However, when more processes are allowed to intervene into lazy joins, the overhead of context switch becomes substantial and makes it slower than lazy joins with two processes. Other than multi-core performance, we also experiment with batch processing, where batch size is fixed to 50. Interestingly, batched lazy join is not significantly slower than eager join, even in training a simple model like LR. We notice that batched lazy join has only slightly worse result than multiprocessing solutions.

Meanwhile, we are curious to know which parts of lazy join play critical roles in the total runtime cost. We decompose a batch into four stages: 1. fetch s - the entity batch, 2 fetch r - the corresponding attribute table entries that will be joined, 3. join s and r, and 4. algorithm computation. The breakdown is illustrated in figure 5. On the synthetic data, we see IO cost dominates the total runtime, as fetching s takes up about 50% of the total runtime and joining a batch costs about 40%. Surprisingly, computation only accounts for a small part of the total runtime.

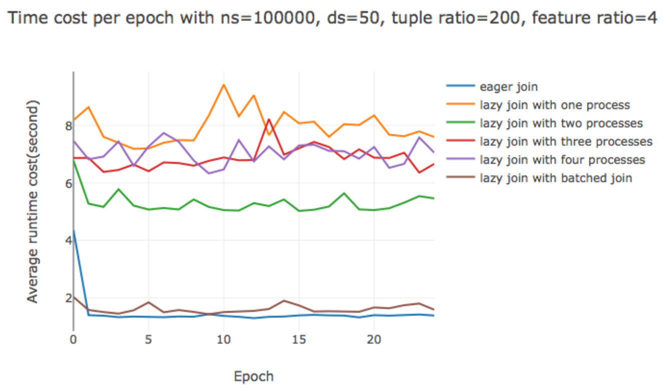

Figure 4: Logistic Regression on Synthetic Data

5.3 Results on Real-world Data

We now present the end-to-end ML training results on the real data. Similar to the experiment on synthetic data, we use logistic regression, naive neural network and the SGD optimizer in Tensorflow. For LR, we present the result comparing batched lazy join and eager join in figure 6. Overall, we notice that eager join is significantly faster than lazy join during training times. On Expedia and Flights, eager join is about 50% faster. On Movie and Yelp, eager join is also about 30% faster.

To figure out why there is a huge gap, we pick Expedia and decompose the runtime in one batch, illustrated in figure 7. This time, each batch is broken down into six steps: 1. fetch s, 2. convert

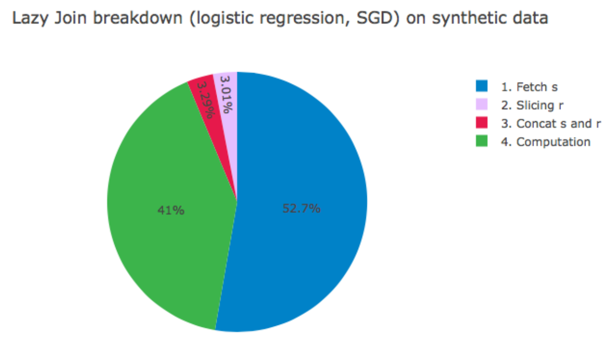

Figure 5: Breakdown on Synthetic Data

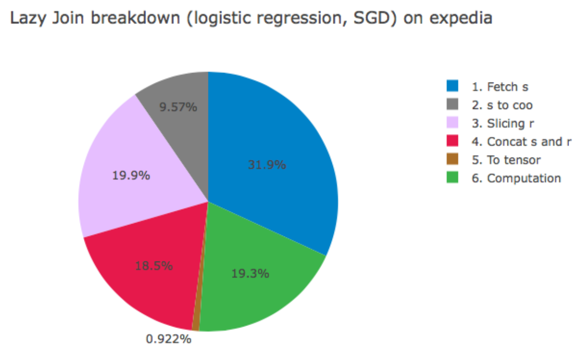

Figure 7: Breakdown on Expedia

s batch to coo format, 3. fetch entries in r that will be joined with s batch, 4. join s and r, 5. convert joined sparse matrix into tensor, 6. computation. Step 2 and 5 are overheads on converting data formats if we intend to consume another data format other than native Tensorfow tensors. Step 1, 3 and 4 are I/O cost at performing lazy joins. At last, only step 6 is the computational cost, which only accounts for about 20% of the total runtime. Overall, we find that the overhead of lazy joins still plays a critical role in adding up the entire runtime.

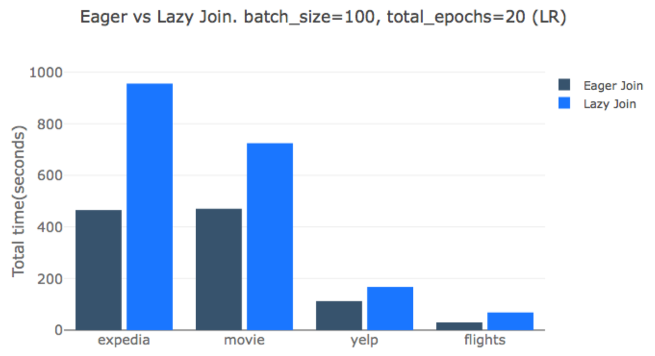

Figure 6: Runtime Cost Comparison (LR)

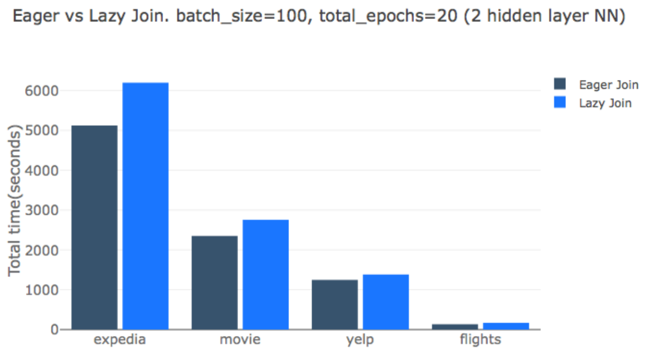

Figure 8: Runtime Cost Comparison (NN)

What about other computation intensive training? We conduct the same experiment on comparing lazy joins and eager joins, except on using a shallow neural network. We illustrate the result in 8. As training become more computation intensive, lazy joins and eager joins tend to converge at the average runtime. On Expedia and Movie, we see about 10% overhead on lazy joins. On Yelp and Flights, the gap disappears as the computation dominates the total runtime.

5.4 Lessons Learned

Regarding the experiment results, we have two major takeaways:

First, the overhead of using multicore on a single join stream might outperform the benefits of using it. Using too many cores will merely put burdens on OS that needs to handle expensive context switches and other hardware resources intervenes.

Second, lazy join definitely suffers from overhead and therefore is slower than eager join. However, the overhead might be trivial if the training process is computation intensive. As shown in our experiment, when training a shallow neural network, lazy joins and eager joins may have very similar runtime cost per epoch. In the world of more complex model training, the total overhead of lazy join should not be a significant concern.

6 CONCLUSIONS

Relational data is ubiquitous in the machine learning world. Scientists are forced to eagerly multiple tables before training models. We extend the simple idea of batched lazy join as a library called MorpheusFlow. MorpheusFlow supports basic data formats widely used in ML such as NumPy matrices and CSV files. With the help of MorpheusFlow, we have shown that lazy joins at training time can be almost as fast as eager joins. We verified that it is safe to use lazy joins in training computation-bounded models. Our work has also prepared us for further exploration on join pattern in deep learning.

All of our codes and the datasets are publicly released on the project webpage: https://adalabucsd.github.io/morpheus.html.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] R. Ricci. E. Eide and C. Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:the magazine of USENIX* 39, 6 (2014), 36–38. https://www.usenix.org/publications/login/dec14/ricci

[3] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (12 2014).

[4] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1969–1984. https://doi.org/10.1145/2723372.2723713

[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113