Database Storage Design for Model Serving Workloads

by

Amitabh Das

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2021 by the
Graduate Supervisory Committee:

Jia Zou, Co-Chair
Ming Zhao, Co-Chair
Yingzhen Yang

ARIZONA STATE UNIVERSITY

August 2021

ABSTRACT

The meteoric rise of Deep Neural Networks (DNN) has led to the development of various Machine Learning (ML) frameworks (e.g., Tensorflow, PyTorch). Every ML framework has a different way of handling DNN models, data types, operations involved, and the internal representations stored on disk or memory. There have been initiatives such as the Open Neural Network Exchange (ONNX) for a more standardized approach to machine learning for better interoperability between the various popular ML frameworks.

Model Serving Platforms (MSP) (e.g., Tensorflow Serving, Clipper) are used for serving DNN models to applications and edge devices. These platforms have gained widespread use for their flexibility in serving DNN models created by various ML frameworks. They also have additional capabilities such as caching, automatic ensembling, and scheduling. However, few of these frameworks focus on optimizing the storage of these DNN models, some of which may take up to $\sim$ 130 GB storage space ("Turing-NLG: A 17-billion-parameter language model by Microsoft" 2020). These MSPs leave it to the ML frameworks for optimizing the DNN model with various model compression techniques, such as quantization and pruning.

This thesis investigates the viability of automatic cross-model compression using traditional deduplication techniques and storage optimizations. Scenarios are identified where different DNN models have shareable model weight parameters. "Chunking" a model into smaller pieces is explored as an approach for deduplication. This thesis also proposes a design for storage in a Relational Database Management System (RDBMS) that allows for automatic cross-model deduplication.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my thesis advisor Dr. Jia Zou, Assistant Professor in the School of Computing, Informatics, and Decision Systems Engineering. While I started working on her research projects, notably the work done on the Lachesis system accepted for publication at VLDB 2021, I gained inspiration and interest in pursuing a topic of my own. Prof. Zou's careful guidance and feedback, not only for my thesis work but also for student life in general, helped me immensely throughout the entire process. I am also grateful to Dr. Ming Zhao and Dr. Yingzhen Yang for their time, continuous valuable feedback, and serving on my committee for my Master's thesis.

I am also grateful to have the opportunity to be a part of the first few students during the inception and naming of Dr. Jia Zou's lab - the Cactus Data-Intensive Systems Lab. With the help and advice of the students at the lab, especially Lixi Zhou and Pratik Barhate, I was able to gather ideas and iron out any issues that I had while working on this thesis.

Finally, I am thankful to my parents and my brother, without whom I would have never had the courage to be here pursuing my Master's degree or the will to keep going. I am also thankful to my roommates who helped me keep my sanity while working on my thesis.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

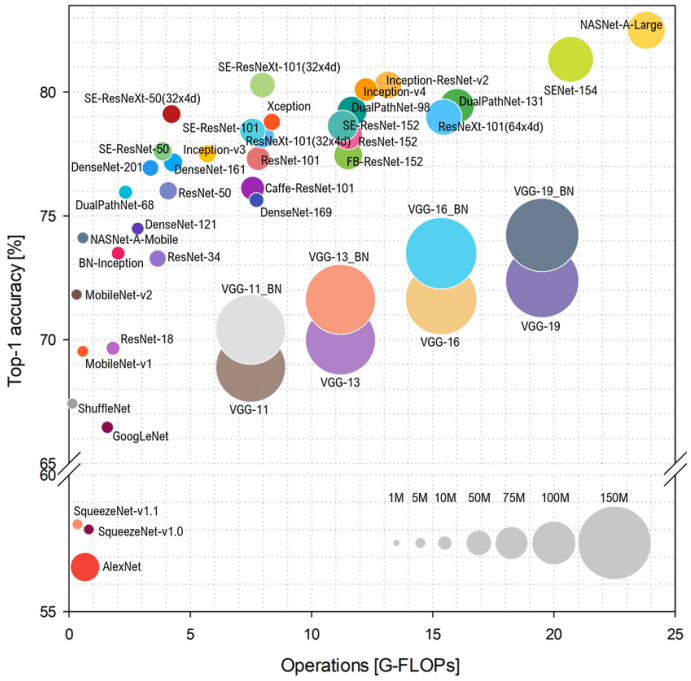


Figure 1: Increasing Model Sizes and their improvement in accuracies (Bianco et al. 2018)

Even though the machine learning community was introduced to the term "Deep Learning" way back in 1986 ("Deep learning" 2021), it has gained widespread use only recently. Large DNN models such as ResNet (K. He et al. 2016) were introduced in 2015. These models are only getting more intricate and more accurate, as seen in Figure 1. One reason for the late surge in popularity is the recent improvements in computer hardware. While Moore's Law ("Moore's law" 2021) saw improvements in general-purpose computer hardware, it is still not powerful enough to train some of the larger and deeper DNN models in reasonable time frames, let alone be used in

real-time scenarios. Figure 2 shows the difference in DNN model loading time and inference time.

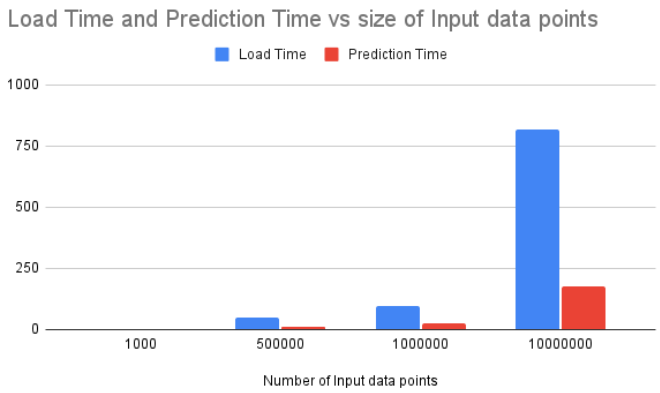


Figure 2: Overhead of reading from a storage medium vs Inference time

The gap in the advancement of DNN models and general-purpose hardware gave rise to special-purpose hardware such as GPUs, TPUs, and FPGAs in the use of Deep Learning. However, the Tesla V100, one of the most advanced GPUs, has 32 GB of memory. A model with over 1.3 billion parameters cannot fit into this GPU, and many such models are used in academic research and the industry. DNN models are increasingly being deployed to eCommerce websites, edge devices, and other user-facing applications. Such deployments are usually handled by Model Serving Platforms (e.g., Clipper, Tensorflow Serving). The problem of storage size and computational cost is compounded due to the deployment of multiple models. Various solutions such as network pruning and model compression have been proposed to get around this issue. However, these techniques are applied to each model individually and require significant fine-tuning to retain their original accuracy (Lee and Nirjon 2020). Model serving platforms store models as provided by the ML frameworks, with or without optimizations like model compression and network pruning. To serve models from

various ML frameworks, such platforms focus on interoperability between these various ML frameworks rather than optimizing the storage of DNN models served by these platforms. There is not much focus on cross-model deduplication.

If, given a set of DNN models, there is a way to share some of the layers between these models or part of the weight parameters between these models, storage of these models can be optimized. Further, loading these shared weights once and using them in subsequent computations can avoid incurring data loading penalty from disk to RAM or GPU.

The contributions of this thesis are as follows:

- Identify the scenarios where layers of DNN models could be similar. Experiments involve DNN models of the same architecture trained on different datasets, ensembles, incrementally trained DNN models, and Knowledge Distillation.
- Evaluate the efficiency of block-based DNN model deduplication. Experiments involve DNN models having the same architecture and different architectures.
- Present and evaluate a design for the storage layer of a Relational Database Management System to automatically deduplicate parts of a DNN model's weight parameters based on blocking or chunking techniques.
- Define the deduplication problem of grouping similar blocks of multiple DNN models as a bin-packing problem.
- Define the partitioning problem of blocks of multiple DNN models in a distributed system to balance deduplication and shuffling overhead.

The rest of the thesis is organized as follows: Chapter 2 briefly introduces the related work. Background information of concepts and systems that are used in this thesis is described in chapter 3. Chapter 4 tries to identify scenarios where cross DNN model similarity can be exploited to optimize DNN model storage. Chapter

5 presents a storage system design for a Relational Database Management System to perform cross DNN model deduplication automatically. Chapter 6 evaluates the proposed design of the storage system. Finally, chapter 7 concludes this thesis and poses two crucial questions to help further improve the design of the storage system for future work.

Chapter 2

LITERATURE SURVEY

2.1 Model Storage in Machine Learning Frameworks

2.1.1 Tensorflow

There are multiple ways to save models created using Tensorflow. `Checkpoints` allow for storing just the weights of the model used in the framework. In Python, one can save the weights of a model by invoking the `save_weights` method of the Model object ("TensorFlow Documentation: Checkpoints" 2021). These are binary Protobuf files that contain the weights of the model and the mapping of the weights to a specific layer of the model ("TensorFlow Source Code Documentation" 2021; "Tensorflow Repository" 2021). First, the model must be created using the framework, and then the `load_weights` method of the model object must be invoked to reuse the weights.

The `SavedModel` format allows storing the entire model, the variables, and the computation, onto the disk. In addition, this format allows for easy sharing or deploying to other compatible frameworks such as TFLite, TensorFlow.js, TensorFlow Serving, or TensorFlow Hub ("TensorFlow Documentation: SavedModel" 2021). On the disk, the `SavedModel` is organized as shown in 2.1.

Listing 2.1: Contents of Tensorflow's SavedModel

```
assets  saved_model.pb  variables
```

The `saved_model.pb` is a binary protobuf file that stores the model architecture and training configuration (including the optimizer, losses, and metrics) required to run the model ("TensorFlow Documentation: Saving Models" 2021). `variables` stores the weights of the model that is saved. `assets` stores other arbitrary files needed by Tensorflow ("TensorFlow Documentation: Asset" 2021).

Protocol Buffers (Protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data ("Google Protobuf Repository" 2021). A protocol buffer message is a series of key-value pairs. The binary version of a message uses the field's number as the key. When a message is encoded, the keys and values are concatenated into a byte stream ("Google Protobuf Documentation: Encoding" 2021).

2.1.2 PyTorch

Similar to TensorFlow, Pytorch allows for storing entire models or just the weights of the model. The weights of a PyTorch model are called the `state_dict`. It is a mapping between a layer and its weight. PyTorch serializes this mapping using Python's pickle module ("PyTorch Documentation: Saving and Loading Models" 2021). Serializing a Python object using pickle converts the object hierarchy into a byte stream. This byte stream can be converted back into an object hierarchy ("Python Documentation: pickle" 2021). Since most Python objects can be serialized using pickle, entire Pytorch models (or even multiple models) can also be serialized and deserialized from the disk. A user can save or load these objects using `Torch.save` and `Torch.load`, respectively.

2.1.3 Open Neural Network Exchange (ONNX)

The interoperability of DNN models is now being given importance due to the wide variety of machine learning frameworks and the numerous models published on the internet by researchers and data scientists ("ONNX" 2021). ONNX is a standard developed for a model's computation graph, operators, and data types. The computation graph is modeled as an acyclic graph, with the nodes as operators that accept one or more inputs and provide one or more outputs of the same or different data types. This graph serves as an intermediate representation (IR). Machine learning frameworks supporting ONNX provide implementations of these operators and the appropriate data types. Underneath, this IR and the data are captured as a serialized protobuf format.

2.1.4 Storage in Model Serving Frameworks

MauveDB (Deshpande and Madden 2006) is built on an RDBMS where the models are defined as a complex materialized view over the underlying data. Tensorflow Serving works with models built in TensorFlow, and so, the storage format of the models is the same as specified in 2.1.1. Other model serving frameworks such as Clipper (Crankshaw et al. 2017), DLHub (Chard et al. 2019), Nexus (Shen et al. 2019) support models created from machine learning frameworks such as Tensorflow and Caffe. They either keep models in isolated containers in their own configured environments or directly on the disk. Therefore, the storage format of the models served is the same as specified by the frameworks that created it. ModelHub (Miao et al. 2017) tries to compact multiple DNN models of the same architecture but

different versions by storing a single float matrix and maintaining deltas (differences) between the versions. vDNN (Rhu et al. 2016) stores the models as is on the disk but reduces the memory usage of the model by exploiting the data dependency specified by the computation graph of the DNN model. Keeping the required weights in memory and discarding the rest reduces the average GPU memory usage of AlexNet by up to 89%, OverFeat by 91%, and GoogLeNet by 95%.

2.2 Model Compression

Over the last few years, DNN models have improved their accuracy significantly. Since the breakthrough of the 2012 ImageNet competition with AlexNet, several other DNNs with increasing complexity have been used to push the last known performance (Canziani, Paszke, and Culurciello 2017). ResNet demonstrated the success of a DNN model with depth 152, previously thought impractical due to vanishing/exploding gradients, winning first place on the ILSVRC 2015 classification task (K. He et al. 2016). The recent advances in specialized hardware such as GPUs, TPUs, FPGA have improved the time required to train such deep networks (Wang, Wei, and Brooks 2019; Guo et al. 2017).

As these models get larger or deeper or both, they incur high memory consumption, computational overhead, and energy costs. For example, the Tesla V100, one of the most advanced GPUs, has 32 GB of memory. A model having over 1.3 billion parameters cannot fit into this GPU. However, DNN models such as Microsoft's Turing-NLG have over 17 billion parameters (Ganesh et al. 2020). Consumer devices such as mobile phones or edge devices do not even compare to GPUs' memory or

computational power, so deploying such models in these devices would be impossible. Such problems have given importance to research in model compression techniques.

2.2.1 Model Compression Techniques

2.2.1.1 Quantization

Reducing the number of bits required to represent the weights of a neural network is called quantization. Generally, 32 or 64-bit representation of floating-point numbers is used in machine learning frameworks such as PyTorch and Tensorflow ("Floating-Point Formats and Deep Learning" 2020). Moving from a 32-bit representation to a 16-bit representation in large models such as the Turing-NLG would result in considerable savings in storage space required and memory consumed. Tensorflow supports Post-training quantization to 8-bit integer, full integer, or 16-bit float (half-float) representations. 8-bit integer makes the models 4x smaller, with 2x-3x speedup ("TensorFlow Lite Documentation" 2021). In the extreme case, 1-bit weights have been used to represent weights of a model, i.e., binary weight neural networks. Such binary representations are directly learned during training. Some examples are BinaryConnect, BinaryNet, and XNOR (Cheng et al. 2017).

2.2.1.2 Network Pruning

Network pruning is another model compression technique that seeks to remove redundant components from a DNN model. These components can be channels or filters in CNN kernels, neurons, or layers. Removal of such components can result

in a lightweight, memory, and computationally efficient model and produces faster inference with minor loss in accuracy. Pruning is performed iteratively till the desired criteria (such as percentage of weights pruned) is met, or the accuracy loss is below a specified threshold (Neill 2020). Usually, the network is trained after pruning the network to fine-tune the model. Network pruning is divided into four categories - Channel pruning, Filter pruning, Connection pruning, and layer pruning (Mishra, Gupta, and Dutta 2020).

2.2.1.3 Low-Rank Approximation

Since DNNs are composed of tensors (matrices are second-order tensors), they can be decomposed into lower-rank approximations that can remove redundancies in the parameters. Insignificant weights can be removed from the tensors, thereby reducing the storage and computational requirements of the DNN model. SVD is a popular approach to decompose the weight tensors (Neill 2020). These low-rank approximations are made layer-by-layer. After decomposing a layer, the layers above are fine-tuned. This approach is straightforward, but the decomposition operation is expensive. The layer-by-layer approach also does not allow for global parameter compression (Cheng et al. 2017).

2.2.1.4 Knowledge Distillation

Knowledge distillation is when the information from a large model A is transferred to a smaller model B. Such transfer of information can be done using the logits output from model A that acts as a soft target for model B or directly training model B on

model A's softmax output (Mishra, Gupta, and Dutta 2020). As a result, model B could be much smaller than model A while maintaining similar or even better accuracy than model A. Knowledge distillation can also be applied in a scenario where multiple models in an ensemble are distilled into a single smaller model. For example, the ensemble of 10 ResNet-28-2 models trained on CIFAR10 with an accuracy of 96.33% can be distilled into a single model of the same architecture to obtain an accuracy of 96.16% (Allen-Zhu and Li 2020).

2.2.2 Machine Learning based Compression

Instead of handcrafted techniques like above, AutoML for Model Compression (AMC) leverages reinforcement learning to automatically find the sparsity in each layer and improve the model compression quality. The entire compression process is automated based on learning-based policies through trial and error and penalizing accuracy loss while encouraging model shrinking and speedup. AMC tries to determine the compression policy for each layer, as they usually have different redundancy (Y. He et al. 2018). With this approach, the authors report up to 60% compression ratio in ResNet50 trained on the CIFAR10 dataset with slightly higher accuracies as a side effect of the pruning.

2.2.3 Other Methods

(Lee and Nirjon 2020) takes a different approach to model compression on edge and embedded devices. Instead of compressing each model separately, they focus on multi-model compression. The authors observe that weights are sparse and that the

consequential weights are in the vicinity of each other. Based on these observations, the paper proposes a technique to preserve the significant weights of multiple DNN models by matching similar weight partials (parts of a weight matrix or tensor) and grouping them to form a single weight partial. Finally, all the DNN models are retrained to optimize the accuracy after sharing the weight partials. Then all of these weights could be loaded into the main memory, and a DNN model can use the required weights to function normally. The experiments presented in the paper packs MobileNet, FaceNet, IM2TXT, Place205, UrbanSound, and GSC into an embedded system, reducing the total memory size required from 604 MB to 146 MB, with little to no accuracy loss. The advantage of this approach is that the context switching overhead between multiple DNNs is reduced drastically, improving multi-model inference performance.

(Vartak et al. 2018) develops a system called MISTIQUE to capture, store, and query model intermediates for model diagnosis. They define model intermediates as the data artifacts related to the model including input data, prediction values, and data representations produced by the model. To improve storage of these intermediates, the paper proposes several quantization techniques suitable for the purpose of model diagnostics. Lower precision floating point representations can be used to gain 2X and 4X reduction in storage with no effect on diagnostic accuracy. Also, the visualization of these neuron activations cannot show >256 shades of the same color. Since many diagnostic techniques are based on relative activations, the values can be represented as quantiles. Further, values can be directly stored as binarized data using a given threshold for diagnostic queries that use an explicit activation threshold. The authors also observe that intermediates in traditional ML pipelines often have many identical columns. Consecutive intermediates often only differ in a handful of features. Predictions from multiple models for the same task, intermediates from

different epochs for the same DNN and quantized versions of intermediates also have significant simialrity. Hence, they propose chunk based similar and exact deduplication techniques to further reduce the storage costs. Identical chunks are detected simply by computing a hash. For detecting similar columns, they use MinHash with Jaccard similarity above a threshold τ. However, the authors only perform exact deduplication for DNN model intermediates, since these are seldom similar.

2.3 Data Deduplication

2.3.1 Growth of Digital Information

Storage requirements have been multiplying since computers became mainstream. The improvements and increase in the production of both commodity and enterprise hardware and devices that capture and create digital data exponentially increase the storage requirement. The forecasts based on the recent growth of IoT hardware indicate that centralized clouds will have a hard time keeping up with the user's data (Koo and Hur 2018). Based on a survey by EMC, different users often store the same or similar data on the cloud, leading to almost 75% of duplication (J. Wu et al. 2019). A recent report by Seagate and IDC predicts the sum of data generated by 2025 to be 175 zettabytes ("Rethink Data: Seagate US" 2020).

As such, researchers have come up with various ways to tackle the problem of storing such vast amounts of data. Data Deduplication is a popular technique, demonstrated to reduce storage costs by more than 50% in standard file systems and by up to 90% to 95% for backup applications (Shin, Koo, and Hur 2017). There are many ways to frame the data deduplication problem. The main goal of deduplication

is to find duplicate instances of a piece of data and store only one such instance to reduce storage requirements. Usually, metadata is also generated as part of the deduplication process to reconstruct the original data. In the field of databases, data deduplication refers to finding approximately similar records within a database. For many database deduplication/record linkage applications, a common approach is to divide the problem into four stages (Borthwick et al. 2020):

1. Normalization
2. Blocking
3. Pairwise Matching, and
4. Graph Partitioning

Other approaches such as machine-learning or crowd-sourcing can also be included to augment the above process (Wang, Xiao, and Lee 2015).

2.3.2 Deduplication in Databases

Dedoop (Kolb, Thor, and Rahm 2012) uses the Map-reduce approach to deduplicating data. This system also allows for using custom matching techniques with machine learning. Dedoop uses a data blocking approach to group records with similar keys and partitions these blocks to reduce tasks based on these keys. Multiple keys are used, and the smallest common blocking key between a pair of records is used to reduce redundant computations. Data skew is handled by splitting large blocks into smaller blocks to utilize all the reduced nodes fully.

With enormous amount of data, (Borthwick et al. 2020) purports that different blocking algorithms would be successful. The authors of this paper utilize Locality Sensitive Hashing (LSH) to build blocking keys for such massive databases. Each

of the records is run through the MinHash algorithm (W. Wu et al. 2020), and the bands act as the blocking key. The size of the blocks is limited to balance precision and recall. The blocks sized less than the limit are deemed acceptable, and every pair in the block are compared. Then, additional co-occurring keys for oversized blocks are searched to get them below the size limit. Not surprisingly, this approach is also amenable to the map-reduce paradigm.

2.3.3 Streaming Data Deduplication

(Bhagwat et al. 2009) tackles data deduplication in a streaming file backup scenario. Chunking the streaming data to remove duplicates results in fine-grained deduplication. Usually, fingerprints are generated for each of these chunks, resulting in an extensive fingerprint index. They achieve improved memory usage by generating two tiers of indexes - one representing the entire file and one for every chunk in the file. The first tier index is used to group similar files. When a new file comes in, the two indexes are generated, and its first-tier index is compared with the other first-tier indexes to find matches. Once a match is found, one can selectively compare with fewer chunks rather than naively comparing with all. This process can also be distributed and parallelized by partitioning the index.

Scaling out such deduplication across a cluster can be challenging due to the need for centralized bookkeeping. (Khan et al. 2018) proposes an approach that involves Distributed Hash Tables (DHT) for a distributed deduplication metadata sharding scheme to complement the architecture of shared-nothing storage systems. DHTs have the remarkable capability to allow continual node arrivals, departures, and failures ("Wikipedia: Distributed hash table" 2021), all with minimum work of re-shuffling

the data. The chunk's fingerprint has a mapping to a node. This method avoids any requirement for a centralized metadata server(s) or any broadcast techniques to add a chunk into the correct node.

2.3.4 Other Deduplication approaches

Combining Machine learning approaches with crowd-sourced feedback is a technique used in many eCommerce websites and entertainment products. Crowd-sourced feedback is helpful in scenarios where the data semantics are unknown to the machine learning algorithm. However, crowd-sourcing is an expensive operation and is prone to errors. The algorithm must take steps to minimize costs. Care must also be taken to reduce the amplification of human error in the clustering process. (Wang, Xiao, and Lee 2015) describes an Adaptive Crowd-Based Deduplication (ACD) that consists of three steps:

1. Pruning phase - this phase removes pairs with similarities lower than a threshold using a machine learning algorithm.
2. Cluster generation phase - An initial clustering is generated using a small number of crowd-sourced tasks.
3. Cluster refinement phase - Based on the previous clustering, a new set of crowd-sourced tasks are generated to return the final deduplicated records.

Chapter 3

BACKGROUND

3.1 A Brief Introduction to Machine Learning

Machine learning (ML) can be seen as an alternative to the engineering design flow. (Simeone 2018a) explains the engineering design flow as a process that conducts an in-depth analysis of the problem domain, capturing the key features of the study into a mathematical model, which is typically done by domain experts. In contrast, the machine learning process involves collecting large data sets, e.g., labeled speech, images, or videos, and using this information to train general-purpose learning machines to carry out a desired task. It could be seen as a faster or less expensive process than the engineering design flow. "General-purpose" in this context is the ability of a machine to perform accurately on new, unseen examples or tasks after having experienced a learning data set. The training examples come from some generally unknown probability distribution (considered representative of the space of occurrences). The machine has to build a general model about this space to produce sufficiently accurate predictions in new cases ("Wikipedia: Machine learning" 2021). The process of training an ML model involves providing training data to an ML algorithm (that is, the learning algorithm). The term ML model refers to the model artifact that is created by the training process ("AWS Documentation: Training ML Models" 2021).

The origin of Machine Learning can be traced back to the 1950s when Arthur Samuel created a program for playing championship-level computer checkers. Due to the small sizes of memory back then, the game used alpha-beta pruning to measure

the chances of each side winning instead of searching every possible outcome of a move. Arthur Samuel is the first person to come up with and popularize the term "machine learning" ("Wikipedia: Machine learning" 2021).

There are usually three approaches to Machine Learning (Simeone 2018b):

1. Supervised learning: The data set used to train the machine contains a mapping between an input and the desired output, and the goal is to learn a mapping between input and output spaces. This form of learning is relatively well-understood at a theoretical level as compared to Unsupervised learning.
2. Unsupervised learning: Unlabelled inputs are provided to the machine with no indication of the desired output. This learning technique generally aims at discovering the properties of the mechanism generating the data. Unsupervised learning can be used to discover hidden patterns in data or for feature learning of the inputs ("Wikipedia: Machine learning" 2021).
3. Reinforcement learning: Desired output for every input is not directly provided in this form of learning. However, unlike unsupervised learning, the desired output is usually feedback provided by the environment only after the machine selects an output for a given input or observation. The feedback indicates the degree to which the output fulfills the goals of the learner.

3.2 A Brief Introduction to Deep Learning and Deep Neural Networks

Deep learning can be considered a sub-branch in the Machine Learning tree. They are primarily based on Artificial Neural Networks (ANN), a computing paradigm inspired by the functioning of the human brain. Generally believed to play a significant role in learning in the brain, the critical characteristic of a neuron is that it can scale a

signal passing through it by a scaling factor. This scaling factor is known as "weight." Furthermore, the brain is believed to learn through changes in the weights of a neuron. Thus, different weights result in different responses to input. (Sze et al. 2017).

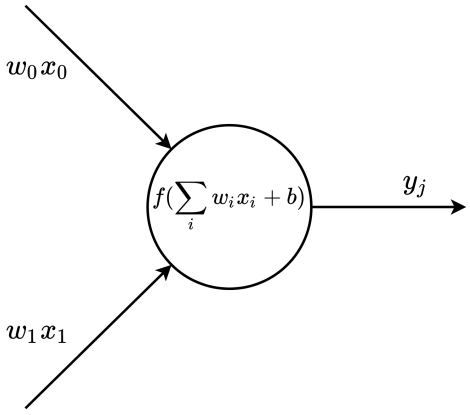


Figure 3: Composition of a neuron (Sze et al. 2017)

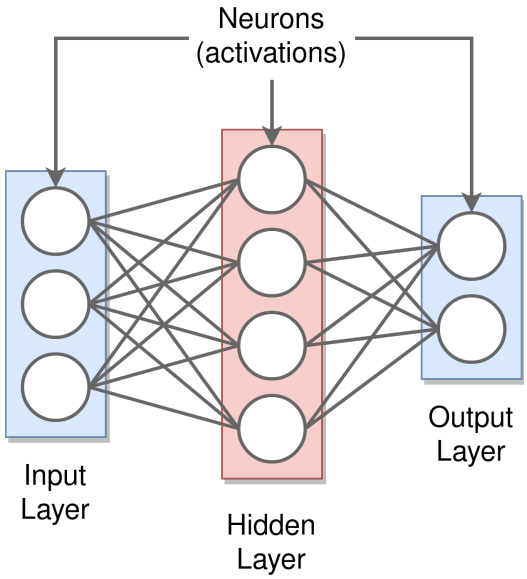


Figure 4: Example of a Simple Neural Network (Sze et al. 2017)

Like the human brain, ANN is composed of many computing cells or 'neurons' that each perform some operation and interacts with each other to make a decision (Arai and Kapoor 2019). The weights associated with these neurons can be updated to "teach" the Neural Network to perform a desired task. ANNs are represented as a set of layers. Each layer consists of several neurons that take inputs from the previous layer's outputs (Guliyev and Ismailov 2016) and performs some operation on these inputs to pass on to the next layer(s). These layers connect each other to form a network, hence the phrase "Neural Network." This network of operations could be modeled as a computation graph.

The word "Deep" in Deep Learning refers to using multiple layers in the network

through which the data is transformed. Thus, it expresses a complicated non-linearity composed of many nonlinear functions, each of which is used to progressively extract higher-level features from the raw input (Fan, Ma, and Zhong 2019). Real-world data sets such as images, audio, and videos have many levels of features of interest to a particular deep learning task. For example, an image is represented by pixel values. The Deep Neural Network (DNN) may group some of these pixel values to form edges and shapes. Then, the DNN may combine these edges and shapes to detect other features such as the nose or eyes. Further, these features may enable the DNN to perform high-level tasks such as face detection. DNNs trained on a data set adjust the weights associated with every layer to extract such features from input data. The resulting set of weights and the structure of the network are referred to as a DNN model. All the weights of a DNN model together are also called the weight parameters of the DNN model. The weight associated with a layer can be represented as a tensor. Usually, this tensor is of rank one or rank two, but it could be higher. Hence, the weight parameters of the DNN model can be thought of as a set of tensors.

More importantly, the deep learning process can minimize or even eliminate the need for manual tuning. This process can determine which features are essential and learn which features to extract at a given layer to reduce redundancy in representation. Such an automatic process is helpful because modeling these features by hand would be extremely hard and or even impossible due to lack of domain knowledge or failure to understand some complex, hidden, or non-intuitive phenomena (Arai and Kapoor 2019).

The very recent success of DNNs can largely be attributed to three factors:

- Large amounts of data that is generated and gathered from various sources (Alam et al. 2020; Wang et al. 2018)

- Huge boosts to both general-purpose and domain-specific hardware.
- Improvement in the accuracy of algorithmic techniques, broadening the domains to which DNNs are being applied (Sze et al. 2017).

However, this success comes at a cost. Models such as Microsoft's Turing NLG contain as much as 17 billion parameters ("Turing-NLG: A 17-billion-parameter language model by Microsoft" 2020) which require considerable memory, storage, and hardware capable of higher Floating-point Operations (FLOPs) to train at faster speeds (Wang, Wei, and Brooks 2019). Techniques such as model compression (Ganesh et al. 2020) have gained popularity to combat these issues.

3.3 Deep Learning Lifecycle and Model Serving

3.3.1 Phases of Deep Learning

3.3.1.1 Data Collection and cleaning

Probably the most time-consuming part of the project is the data collection and cleaning phase. Although Deep Neural Network models are generally resilient to some amount of noise, the quality of the data affects the performance of the model and the downstream applications of the model (Li et al. 2021). Surveys show that upwards of 80% of analysis time is spent during data preparation and cleaning (Krishnan and Wu 2019). This phase usually involves cleaning both unstructured and structured data, enforcing integrity constraints, denial constraints, and functional dependencies, removing duplicates, and normalizing the data from different sources into agreeable ranges (Tae et al. 2019)

3.3.1.2 Training the DNN model

Training is the process of building a model from the data collected (e.g., movie ratings), using a learning algorithm, and optimizing an objective function. Training a DNN model is a computationally-intensive process, requiring multiple passes over the layers of a DNN to adjust weights at each iteration to minimize the objective function. Many of the recent models researched and developed require vast amounts of computational power and memory. The trend is towards larger models, leading to better performance (Ganesh et al. 2020). Today, many deep learning frameworks enable a researcher or a professional to focus on building the model instead of the finer details of implementation (e.g., Caffe, Tensorflow, MXNet). Specialized hardware such as GPUs and TPUs may be used to train models in reasonable times (Wang, Wei, and Brooks 2019). Using cloud platforms (e.g., Google Colab) is another approach to training models, especially useful during the current shortage of GPUs ("GPU Shortage" 2020).

3.3.1.3 Model Inference

The inference is the process of evaluating a model to render predictions on a given input (e.g., predict a user's rating for a movie). Training a model is computationally expensive, requiring multiple passes over large data sets, whereas inference is relatively inexpensive (Crankshaw et al. 2017). One pass over the layers of a DNN model is necessary to produce an inference, eliminating the need for extra memory space for intermediate tensors. Cloud services have gained popularity as more and more models are being deployed in websites accessible by end-users. Their pay-as-you-go

pricing and flexible hardware allocations are more attractive than making a significant investment in hardware (Fang et al. 2021).

3.3.2 Model Serving Platforms

Model serving or Inference serving is becoming more commonplace in user-facing applications and is expected to keep growing. Facebook, for instance, serves tens-of-trillions of inference queries per day (Romero et al. 2020). Just as abstractions over hardware and implementation details were needed to simplify real-world use of machine learning system (Abadi et al. 2016), a solution was required for deploying models in applications which demanded real-time, accurate, and robust predictions under heavy query load (Crankshaw et al. 2017). Not only that but subsequent retraining to adjust the model to the continuous flow of new data and redeployment needed to be handled in a simplified way (Klabjan and Zhu 2020).

Several Model Serving solutions exist, such as Tensorflow serving and Nvidia's TensorRT Inference Server. Cloud providers already offer model serving options such as AWS SageMaker ("AWS SageMaker" 2021) and Google Vertex AI ("Vertex AI" 2021). Recently, server-less computing, whose advantages include high elasticity and fine-grained cost model, brings another possibility for model serving (Y. Wu et al. 2021; Ishakian, Muthusamy, and Slominski 2018). These systems typically expose endpoints such as REST, gRPC, or client API interface, which can query a model. In systems that support auto-scaling and load-balancing, middleware manages provisioning and acts as a single endpoint that routes requests to individual model serving. (LeMay, Li, and Guo 2020)

3.4 Databases and DBMS

3.4.1 DBMS and RDBMS

A database refers to a set of related data and the way it is organized. This data can be accessed via database management systems (DBMS). DBMS is software that allows end-users, applications, and the data itself to interact with the data stored. Given the close relationship between a database and DBMS, the two terms are usually informally used interchangeably ("Wikipedia: Database" 2021). DBMS are classified based on the organization of the data (e.g., Relational, Object-Oriented, XML), the interface provided (SQL, XQuery), or the systems on which they run (distributed cluster, standalone, mobile phones). The core functionality provided by a DBMS is the storage, retrieval, and update of data ("Wikipedia: Relational Database" 2021).

Relational databases (RDBMS) refer to data organization into tables (or relations) of rows and columns. Each row contains a record (or tuple). The columns are also known as the attributes of the record. Generally, a table or a relation refers to a real-world entity (e.g., Customer, Employee), and each record in the table would describe unique instances of that entity ("Wikipedia: Relational Database" 2021). Each table has a key or an index that allows any record to be uniquely identified. Other tables may refer to this key to link each related record (Padhy, Patra, and Satapathy 2011).

The storage of the database is the physical materialization of the data stored. This internal representation is dependent on the implementation of the database, usually the responsibility of the storage engine. It also contains the required metadata to

reconstruct the conceptual representation known to the applications or end-users accessing the DBMS.

An old, popular storage format is the row-storage format. In this type of storage format, the storage engine stores each record of the table as a sequence of rows in contiguous space on the storage medium. All queries directed toward the storage engine are processed in a row-oriented manner. The storage engine must read the entire record from the storage medium to access even a single attribute. Queries that target specific attributes of the records in a table may suffer from overheads due to such contiguous ordering on the storage medium (Chaalal, Hamdani, and Belbachir 2020).

3.4.2 Buffer Pool

Databases employ a self-managed memory environment called the buffer manager. DBMS makes calls to the buffer manager to request pages (for reading or writing). The buffer manager, in turn, interacts with a disk manager that manages the interface to the physical storage devices. When a page is requested, the buffer manager first checks the buffer pool, and if the requested page is found, the page is returned to the caller. Otherwise, if there is a free frame in the buffer pool, the buffer manager requests the disk manager to read the desired page (from disk) into that free buffer frame. If there is no free frame, the buffer manager picks a victim buffer page based on a replacement policy and evicts that victim page. If the evicted page is dirty, then the evicted page is written to the disk before its frame is reused. The disk manager handles read and write requests passed to it from the buffer manager by issuing asynchronous I/Os, to maximize both CPU and I/O resources (Do et al. 2011).

Potentially, a database can work with large amounts of data exceeding far beyond the capacity of the DRAMs.

3.5 Tensor Relational Algebra

3.5.1 Tensors

A tensor is a generalization of a matrix, and the tensor dimension is called its rank. For example, a one-dimensional vector can be called a tensor of rank 1, and a two-dimensional matrix is a tensor of rank two. A single number could be called a tensor of rank zero. ("Wolfram MathWorld: Tensor Rank" 2014).

Any rank 2 tensor can be represented as a matrix, but not every matrix is a rank 2 tensor. The numerical values of a tensor's matrix representation depend on what transformation rules have been applied to the entire system.

3.5.2 Relational Algebra

Relational algebra is a theory that uses algebraic operators to transform one or more relations provided as inputs into an output relation. These operators can be combined to operate on one or multiple relations in a relational database to produce the desired output relation. Unary Relational algebraic operators operate on a single relation (e.g., select, project). Binary relational algebraic operators operate on two input relations (e.g., join, union, intersect) ("Wikipedia: Relational algebra" 2021).

3.5.3 Tensor Relational Algebra

(Yuan et al. 2021) presents Tensor relations as a simple binary relation between keys and multi-dimensional arrays. They include several reasons supporting this notation. For instance, the "chunking" of tensors and mapping the chunks to keys make this a set-based abstraction, hence easily parallelizable similar to relational algebra. They also prove that this approach to tensor relation representation is as powerful as Einstein's notation. Representing tensors as tensor relations allows us to use relational algebra operators on these relations to perform tensor operations.

Tensors allow us to naturally represent multi-dimensional data (e.g., time-series data, image data, video). Tensor operations, such as tensor decomposition, support the analysis of multi-dimensional data, which helps analyze the data. Relational algebra enables semantic manipulation of data using relational operators, such as projection, selection, join, and set operators, such as union and intersection. Combining the tensor and relational models enables it to support data management and analysis operations (Kim 2014).

3.6 User Defined Functions

User Defined Functions (UDF) of any system, as the name implies, are functions or extensions written by the user of that system. UDFs provide the user with a mechanism to extend the functionality of a system for specific use cases. A typical example is a routine written in SQL for a relational database management system.

Today, many RDBMS allows users to write a UDF in SQL and other languages such as C, C++. These provide a way for the user to implement complex functions

while achieving code modularity and reuse. The declarative way of writing functions may complicate some business logic. For instance, ML algorithms are recursive and translate to complex, hard-to-maintain declarative forms (Jankov et al. 2020). Popular systems such as Spark and Hive also have support for UDFs.

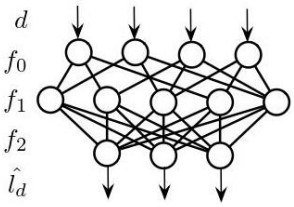
Even though UDFs encourage good programming practices and are a powerful abstraction in an RDBMS, the benefits come at the price of performance penalties. The root cause of the poor performance is the impedance mismatch between the two programming paradigms - the declarative paradigm of SQL and the imperative paradigm of procedural code (Ramachandra et al. 2017).

3.7 Pangea

The ideas presented in this thesis are implemented on a Monolithic Distributed Storage for Data Analytics called Pangea (Zou, Iyengar, and Jermaine, n.d.). Pangea allows storing user-defined objects in C++. A named, unordered collection of a single type of user-defined object can be stored directly into files on the storage medium with no serialization or deserialization overhead (the representation of the object in the memory and the storage is the same). Pangea also provides computations equivalent to the core operators in relational algebra, which the user can extend to implement additional functionality similar to UDFs. Tensor relational algebra can be easily expressed using one or a combination of the computations provided by Pangea. These computations operate on the unordered collection of objects sequentially.

Chapter 4

PROBLEM IDENTIFICATION



$$f_0 = \sigma_0(W_0 d + b_0) \qquad d \in \mathbb{D}$$
$$f_i = \sigma_i(W_i f_{i-1} + b_i) \qquad 0 < i \leq n$$
$$L(f_n, l_d) \qquad l_d \in \mathbb{O}$$

Figure 5: Components of a DNN model (Miao et al. 2017)

A deep neural network model contains two types of components (Figure 5) - learned weights (W_i and b_i) and activation functions σ_i for every layer i. A DNN can contain many layers, each of which can have weights or activation functions or both. These weights are generally tensors (or n-dimensional matrices). In the first layer of any DNN model, the learned weights and activation functions are applied to the input data. The output of the first layer is then fed to the subsequent layers, each layer extracting (activating) the desired features from the input data based on the learning process that created the weights of every layer.

4.1 Model deduplication opportunity

Depending on how the DNN is modeled, it can have a large number (and size) of weights. For example, VGG16 secured the first rank in the ILSVR(Imagenet) 2014 challenge and is widely used in image classification and transfer learning use cases.

It contains about 138,357,544 trainable parameters. This model requires at least 1106860352 bytes ($\sim$ 1.1 GB) of disk space to store, or memory to load, considering each weight parameter as a `double` type. The Microsoft Turing-NLG contains 17 billion parameters and requires over 130GB of storage space. Storing and serving of such DNN models can benefit from the deduplication of such large weights. Of course, two large weights (tensors) are not usually similar to each other (except in certain scenarios such as transfer learning). Given the weight distribution of popular DNN models as in Figure 6, there could be a smaller section within a tensor that could be similar to other sections within other tensors or within itself. If there were similar tensor subsections, the weight parameters corresponding to these tensor subsections could be deduplicated.

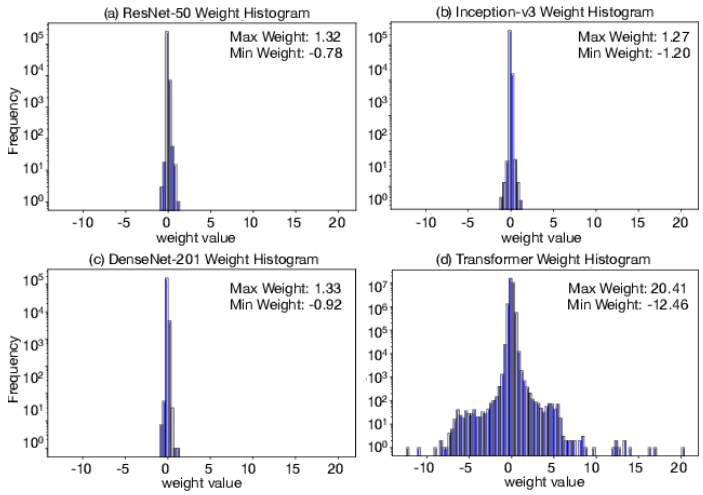


Figure 6: Histogram of the weight distribution for (a) ResNet-50, (b) Inception-v3, (c) DenseNet-201 and (d) Transformer whose weight values are more than 10×higher than the maximum weight value from popular CNNs. (Tambe et al. 2019)

Since DNN model weights are usually made up of real numbers (floating point numbers), finding an exact match is not usually feasible as the domain of the numbers that occur in the weight parameters would be too high. Using low precision for

training is difficult since the learning rate is usually set to a small floating-point value, and the gradient vanishes at lower precisions. However, it has been shown that a DNN model does not usually require high precision floating point numbers for inference (Gupta et al. 2015). The additional precision has no extra benefit and only includes the drawbacks of being computationally intensive, power inefficient, and consuming high memory. In fact, in some cases, DNN models can reach higher accuracy with lower precision (Choi, El-Khamy, and Lee 2020).

The possibility of maintaining inference accuracy with lower precision weight parameters implies that the weight parameters can be rounded off or quantized to potentially improve the chances of finding similar tensor subsections for deduplication.

4.2 Layer Similarity Experiments

The experiments in this section explore a few scenarios that could be amenable to weight parameter deduplication. These experiments compare models layer-by-layer to check their similarity for a particular floating-point round-off (ϵ).

Let $T_1^{\mu_1 \cdots \mu_p}$ and $T_2^{\mu_1 \cdots \mu_p}$ be the tensors of the two layers being compared. Let $n = \mu_1 \times \cdots \times \mu_p$. Then the similarity ($\theta$) between the two layers for a floating point threshold ϵ is computed as:

$$\theta(T_1^{\mu_1 \cdots \mu_p}, T_2^{\mu_1 \cdots \mu_p}) = \frac{\sum_{i=1}^{n} [\left| T_1^{\mu_1 \cdots \mu_p} - T_2^{\mu_1 \cdots \mu_p} \right| \leq \epsilon]}{n} \tag{4.1}$$

In the following experiments, the similarity between two models with similar network structures is tested by comparing the similarity of their corresponding layers. Say two models, model A and model B, are compared. Each layer of model A is compared to the corresponding layer of model B. Since the models compared have the same network, the weights of each layer in model A have the same tensor dimensions

as that of the corresponding layer in model B. Having the same tensor dimensions allows us to perform the difference operation between the two tensors of those layers.

4.2.1 Ensembles

(Allen-Zhu and Li 2020) conducts experiments on models with the same network, trained on the same datasets. They found that the ensembles of such models could have better accuracy than the best accuracy of the individual models. The accuracy boost of the ensemble implies that each of the models can learn features that the others may not. The paper says that each model in an ensemble performs hierarchical feature learning — Despite having different random initializations, each model is still capable of learning the same set of features as the others. Here lies the possibility of weight similarity between the models in the ensemble.

4.2.1.1 Experiment 1a. Ensemble SGDW

An Ensemble of three wide ResNet models with depth 28 and width 2 (ResNet-28-2) are trained on the CIFAR-10 dataset. All have a different random initialization. The optimizer used for training is SGDW (Loshchilov and Hutter 2017).

Let the three models in the ensemble be Model 1, Model 2, and Model 3. Model 1 achieves 75.67% accuracy, Model 2 achieves 76.75% accuracy, and Model 3 achieves 67.41% accuracy. Their ensemble has 78.44% accuracy.

Figure 7 shows the layer similarity between Model 1 and Model 2. Figure 8 indicates the layer similarity between Model 2 and Model 3. Figure 9 shows the layer similarity between Model 1 and Model 3.

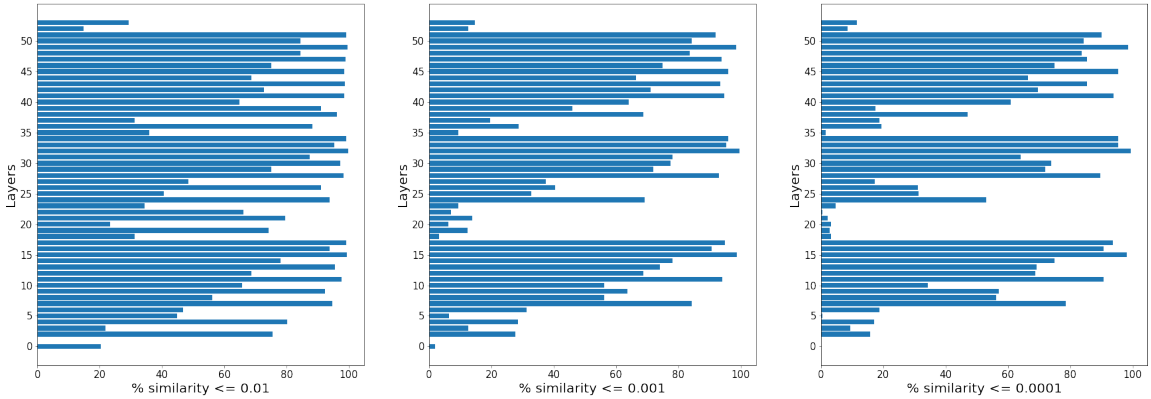


Figure 7: Layer similarity between Resnet-28-2 Model 1 and Model 2 trained with SGDW

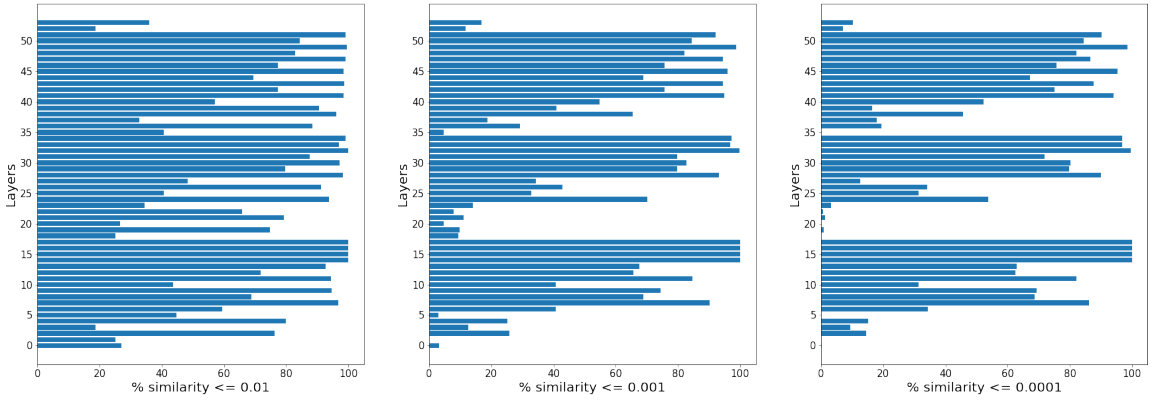


Figure 8: Layer similarity between Resnet-28-2 Model 2 and Model 3 trained with SGDW

In each of these cases, a significant similarity between the layers of the models is observed. As the epsilon floating-point threshold is relaxed, a higher similarity is seen. The pattern of similarity between every pair of models appears to be similar too, which could be exploited for cross-model deduplication.

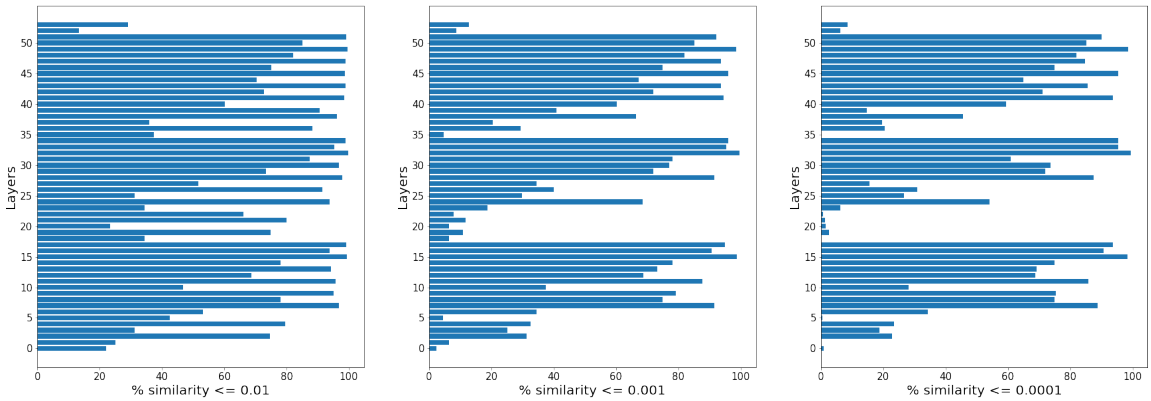


Figure 9: Layer similarity between Resnet-28-2 Model 1 and Model 3 trained with SGDW

4.2.1.2 Experiment 1b. Ensemble SGD

This experiment is precisely the same as experiment in 4.2.1.1, except that the models are trained with the SGD optimizer instead of the SGDW optimizer.

Let the three models in the ensemble be Model 1, Model 2, and Model 3. Model 1 achieves 87.24% accuracy, Model 2 achieves 87.10% accuracy, and Model 3 achieves 87.40% accuracy. Their ensemble has 89.27% accuracy.

Figure 10 shows the layer similarity between Model 1 and Model 2. Figure 11 indicates the layer similarity between Model 2 and Model 3. Figure 12 shows the layer similarity between Model 1 and Model 3.

In each of these cases, a significant difference between the layers of the models is observed. Despite achieving a similar accuracy boost in the ensemble as the previous experiment, very few weights are similar. So, even if the architectures of the models are the same and trained on the same dataset, the similarity between models may vary depending on the way the models are trained.

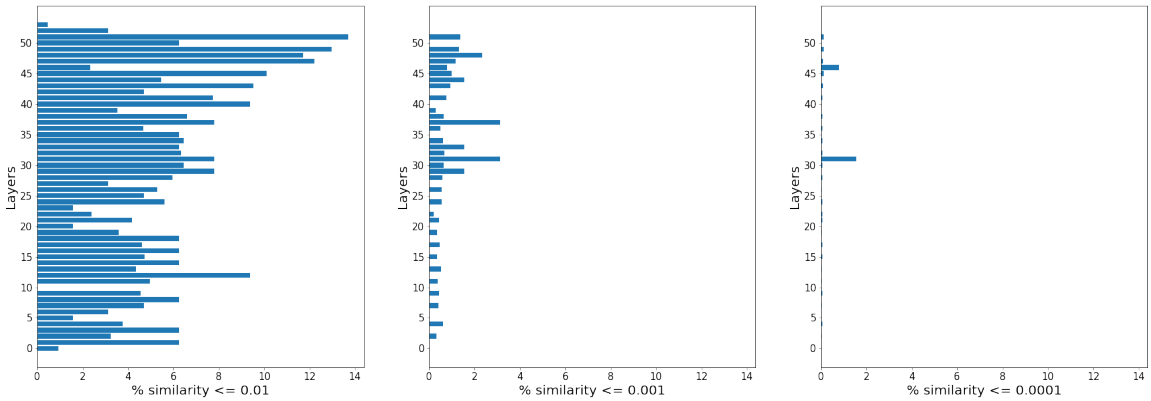


Figure 10: Layer similarity between Resnet-28-2 Model 1 and Model 2 trained with SGD

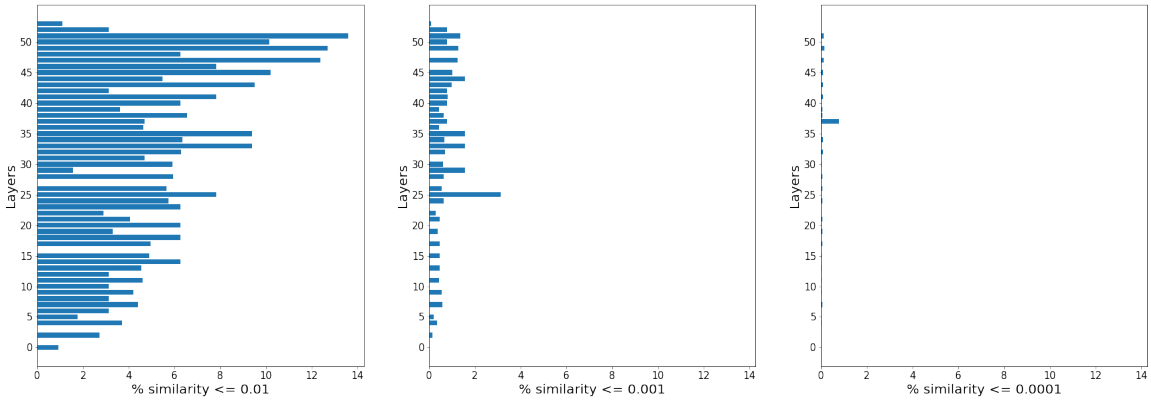


Figure 11: Layer similarity between Resnet-28-2 Model 2 and Model 3 trained with SGD

4.2.2 Teacher-Student Models

4.2.2.1 Experiment 2a. Knowledge Distillation

Two CNN models, Model A and Model B, are created with the same network structure and trained on the MNIST dataset. One of these models acts as a teacher model for the other. Knowledge is distilled from Model A to Model B. This experiment initializes both models with the same random initialization. First, Model A is trained

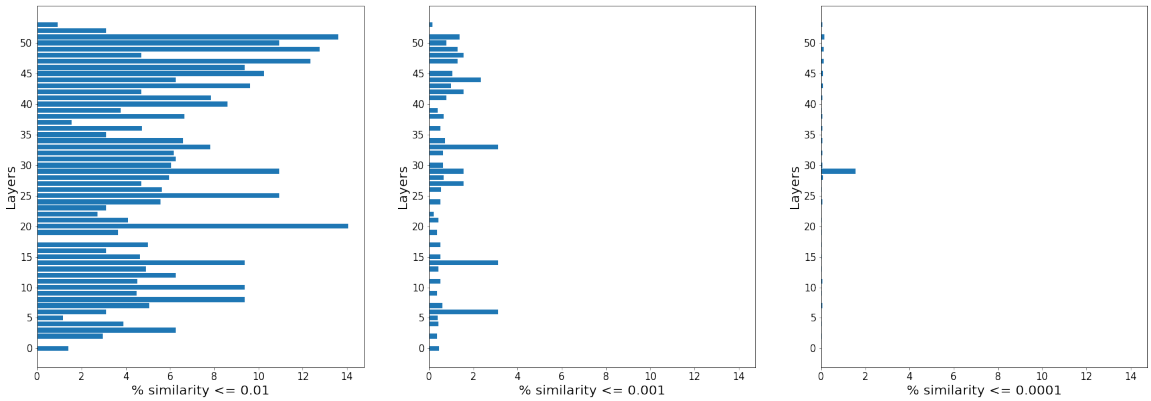


Figure 12: Layer similarity between Resnet-28-2 Model 1 and Model 3 trained with SGD

on MNIST data and labels. Then Model B is trained on MNIST data and the labels generated by Model A. Then, the similarity between the corresponding layers of both the models is measured at every epoch of training, as shown in Figure 13. As seen in the Figure, the difference between the models' corresponding layers grows for a few epochs and does not change any further. This similarity presents an opportunity for cross-model deduplication in the knowledge distillation scenario.

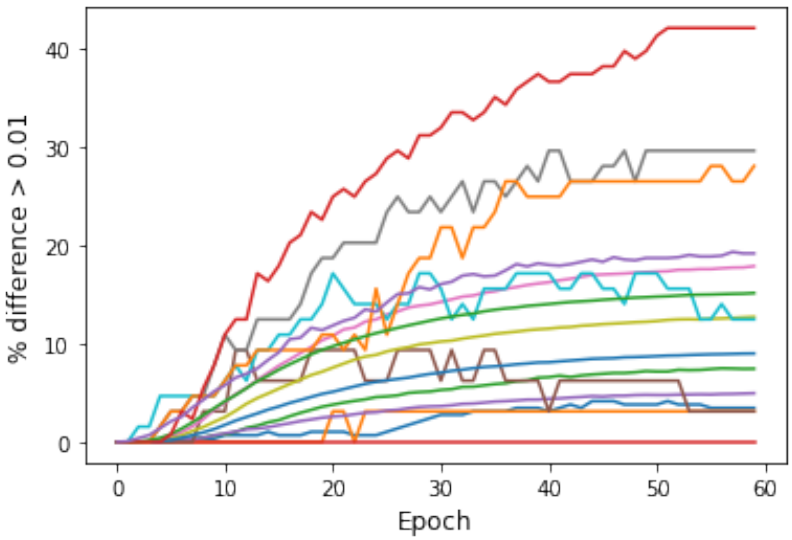


Figure 13: Layer difference between Teacher and Student CNN models at every epoch

4.2.3 Model versioning/Data Drift

Data Drift describes the phenomenon of changing data over time in data streams. This phenomenon may occur when a model is trained and then deployed to serve predictions. Once an initial model is well-trained on historical data, it is then periodically fine-tuned or retrained based on a continuous flow of new data for inference in model serving. New data may be collected every second, day, or week (Klabjan and Zhu 2020). Such kinds of Data drift are classified into a) Sudden Drift, b) Incremental or gradual drift, c) Reoccurring drift (Lu et al. 2018). Maintaining multiple versions of a model to respond to a particular scenario may be helpful in these cases. If multiple models are supported to respond to a data drift that may reoccur, deduplicating these models could be beneficial.

4.2.3.1 Experiment 3a. Sudden Drift

A CNN model, Model A, is created and trained on MNIST. It achieves $\sim 99.5\%$ accuracy on this dataset. Then a sudden drift is emulated by adding noise to the MNIST dataset. Model A now achieves an accuracy of ~ 21.25 on the noisy MNIST dataset. Two scenarios are considered -

1. A new model, Model B, is trained with the original MNIST dataset and the noisy MNIST dataset.
2. Model A is trained on the noisy MNIST dataset to create a new model, Model B.

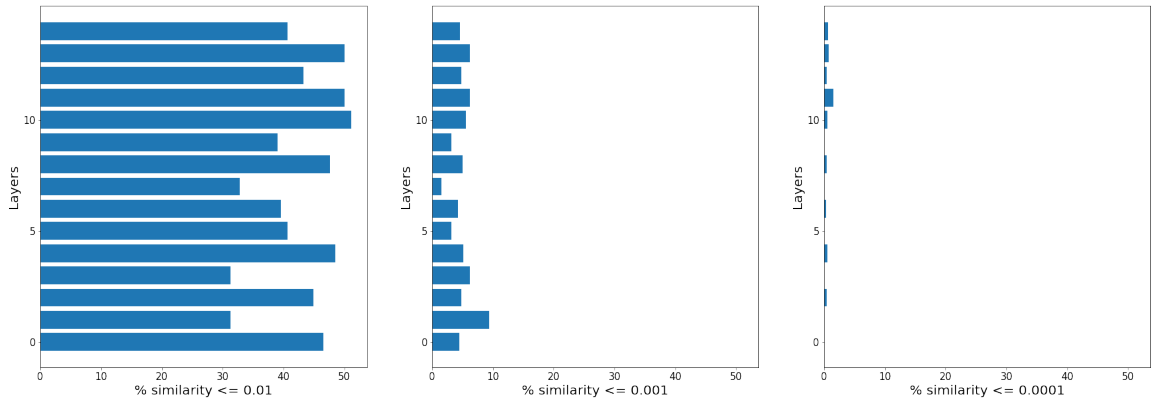


Figure 14: Layer similarity between incremental CNN models versions due to sudden Data Drift

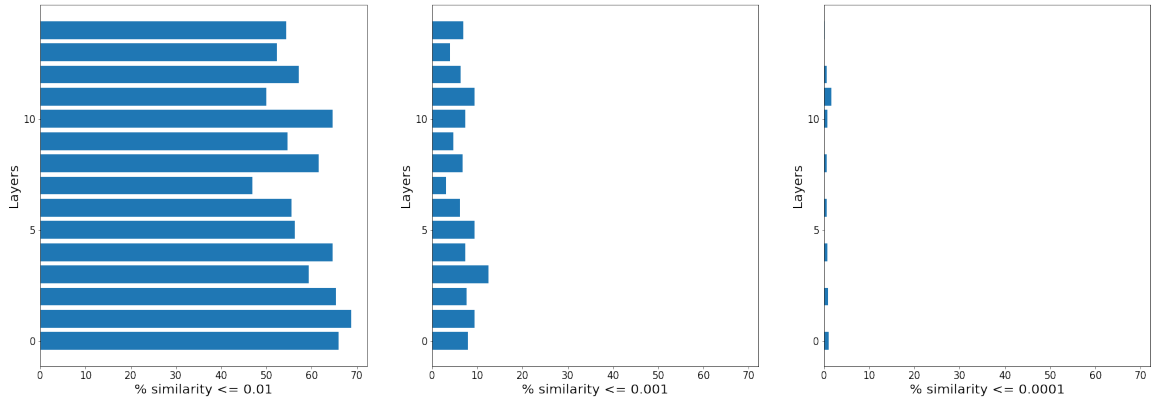


Figure 15: Layer similarity between different CNN models versions due to sudden Data Drift

Figure 14 and 15 shows the similarity between Model A and Model B at various floating-point thresholds for both the scenarios.

4.2.3.2 Experiment 3b. Gradual Drift

A CNN model, Model A, is created and trained on MNIST. It achieves $\sim 99.5\%$ accuracy on this dataset. Then a gradual drift is emulated by adding noise to half the MNIST dataset with a gradually increasing noise level multiplier. A new model,

model B, is trained on this modified dataset. Figure 16 shows the similarity between Model A and Model B at various floating-point thresholds.

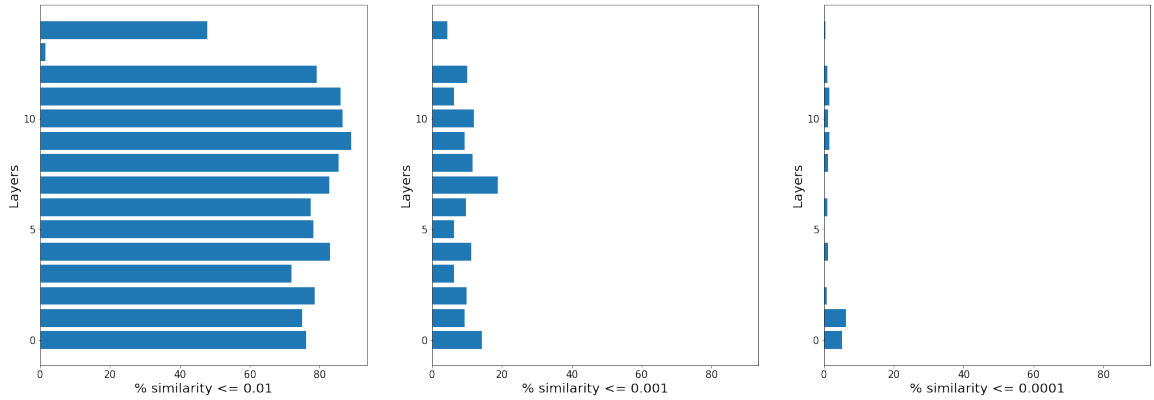


Figure 16: Layer similarity between incremental CNN models versions due to gradual Data Drift

4.2.4 Private and Shared Data

4.2.4.1 Experiment 4a.

When many models $m_1, m_2, \ldots, m_n$ are trained with some common data and with some other data $p_1, p_2, \ldots, p_n$ for each corresponding model, Figure 17 shows the percentage difference between two CNN models trained separately on private subsets of MNIST data and a common subset of MNIST data. The resulting models have similar weights. These overlapping weights could be deduplicated.

	Model 1 vs 2	Model 1 vs 3	Model 2 vs 3
Layer 1	51 / 1024 (4.98046875%)	44 / 1024 (4.296875%)	72 / 1024 (7.03125%)
Layer 2	11 / 8192 (0.13427734375%)	29 / 8192 (0.35400390625%)	85 / 8192 (1.03759765625%)
Layer 3	43 / 16384 (0.262451171875%)	21 / 16384 (0.128173828125%)	105 / 16384 (0.640869140625%)
Layer 4	35 / 320 (10.9375%)	13 / 320 (4.0625%)	52 / 320 (16.25%)

Table 1: Difference at every layer of three CNN models trained on MNIST dataset split 50:50 among shared and private subsets

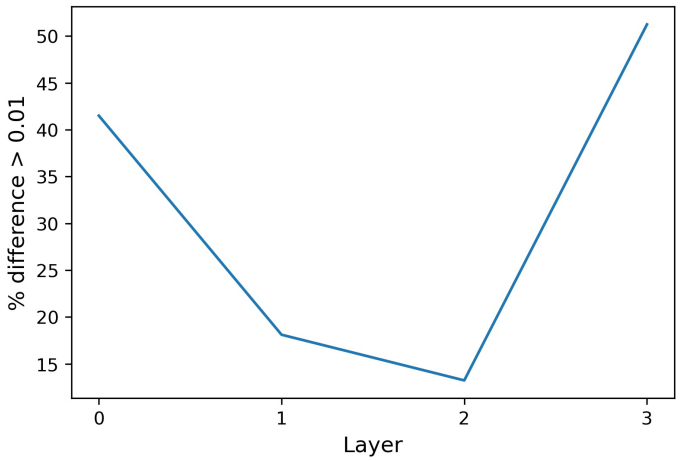


Figure 17: Percentage difference between CNN models trained on MNIST dataset split 25:75 among shared and private subsets

4.2.5 Transfer Learning

In this scenario, multiple models share the weights of specific layers. These shared layers are commonly used as feature extraction layers. The other layers are specialized for a particular task, such as classification between different car brands ("Keras documentation: Transfer learning & fine-tuning" 2021). No experiments are required in this scenario. The weights of the shared layers could be deduplicated in their entirety to save disk space and memory usage on inference.

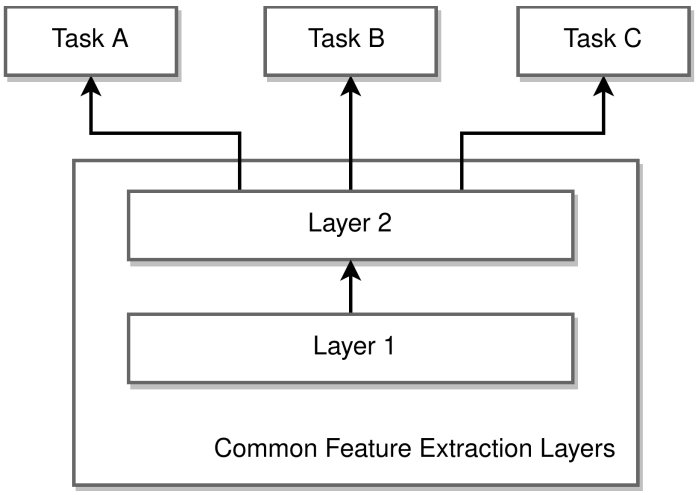


Figure 18: The Transfer Learning scenario

4.3 Block Similarity Experiments

As seen in the experiments conducted in 4.2, there are possibilities of sharing weight parameters between multiple models. This section explores the possible approaches to deduplicating model weight parameters. Considering every single value in a tensor as a target for deduplication is not very feasible. Even a minimal amount of metadata required to keep track of the positions of every value in a tensor could be counterproductive to the goal of weight parameter deduplication.

4.3.1 The blocking approach

Usually, data deduplication is done at a chunk or a block level. A common approach is to split the data to be deduplicated into fixed-sized chunks or blocks and generate fingerprints. These fingerprints are stored in an index to be later matched with a duplicate chunk that arrives for storage. The amount of storage savings gained with

the deduplication approach depends on how data overlaps, the granularity of the chunk sizes, and the method of chunking. In the traditional applications of data deduplication, smaller chunks generally yield better results (Bhagwat et al. 2009).

This section examines block-level deduplication of models with weight parameters represented as tensors of rank 2 (a matrix). A DNN model's weight parameters can be blocked and deduplicated using the following approach: A large matrix could be split into multiple equal-sized matrix blocks. Padding can be applied to the matrix to obtain an even blocking. Once all the matrices associated with every layer of the DNN model have been split into smaller blocks, the duplicate matrix blocks could be eliminated from storage, and metadata for a model could be maintained to refer to the unique matrix block instances. "Blocking a DNN model" and "blocking a DNN model's weight parameters" are used interchangeably in this thesis.

4.3.2 DNN operations on blocked DNN models

DNNs are networks of dense linear algebra operations. A DNN model can be represented as a graph of computations on its weight parameters. This computation graph also defines the data dependency between the weight parameters of a DNN model (Zhou et al. 2020; Yu, Mazaheri, and Jannesari 2020). Standard DNN computations such as dense layer (i.e., matrix multiplication) on a large matrix could be implemented to operate on a set of matrix blocks that together represent the original matrix (Schatz, Van de Geijn, and Poulson 2016). So, there need not be any "translation" from weight parameter representation at the storage level to compute.

As a bonus, blocking of the DNN model's weight parameters is, in effect, a way to partition the model. This partitioning allows us to take advantage of model

parallelism or data parallelism or both (Yuan et al. 2021; Qararyah et al. 2021). A matrix representing the weights of a layer could be partitioned and distributed across a set of compute nodes in a cluster to perform the distributed version of a DNN computation. Such a technique could alleviate the intense memory and compute requirements demanded by today's state-of-the-art DNN models.

4.3.3 Detecting similar blocks

Similar to the experiments in 4.2, two equal sized matrix blocks containing floating point numbers can be compared to check their similarity. The similarity (θ) between two matrix blocks B_1 and B_2, having the same dimensions $m \times n$, under a particular floating-point round-off (ϵ) is defined as:

$$\theta(B_1, B_2) = \frac{\sum_{i=1}^{m \times n} [|B_1 - B_2| \leq \epsilon]}{m \times n} \tag{4.2}$$

In this experiment, two models trained on a dataset are blocked into a set of square matrix blocks. For every block, a list of similar blocks for various ϵ and θ thresholds is compiled by performing pairwise comparisons. Then the similar blocks are removed, and the original models are reconstructed with only the unique blocks. These two models are re-evaluated on the same dataset to check the change in accuracy due to deduplication.

4.3.4 Deduplicating VGG16 and VGG19 models

The block deduplication experiment is performed on VGG16 and VGG19 models at two ϵ thresholds 0.01 and 0.001, and θ thresholds 0.7, 0.8 and 0.9. These models

are blocked into square matrices of dimensions 200×200 to 1900×1900, and the deduplication results at each of these dimensions are recorded. Figures 19 and 20 show the total size (MB) saved and accuracy loss of the models at all the similarity percentage thresholds at ϵ thresholds 0.01 and 0.001 respectively. Darker circles show lower accuracy in VGG19. Larger circles denote lower accuracy in VGG16.

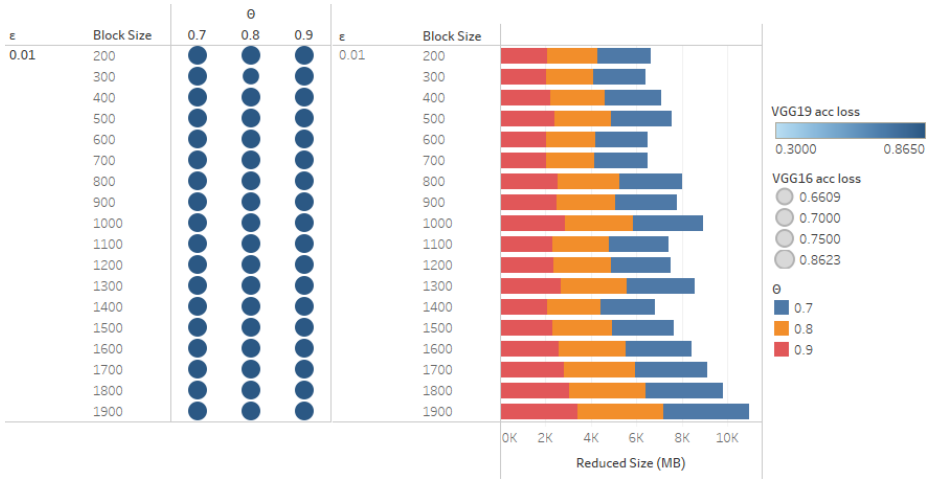


Figure 19: Storage size saved and accuracy loss with various similarity, deduplicating VGG16 and VGG19 models at $\epsilon = 0.01$. $\epsilon = 0.01$ is not a good enough threshold to deduplicate blocks between VGG16 and VGG19.

Based on these results, the ϵ threshold 0.01 is not a good measure to gauge the similarity between these blocks. All the accuracy of the model is lost here. However, ϵ threshold 0.001 deduplicates $\sim$ 825 MB ($\sim$ 27%) with almost no accuracy loss at block size 1300×1300 and $\theta \geq 0.7$.

4.3.5 Deduplicating Two CNN models trained on MNIST

The block deduplication experiment is run with two CNN models at two ϵ thresholds 0.01 and 0.001, and θ thresholds 0.7, 0.8 and 0.9. These models are blocked into

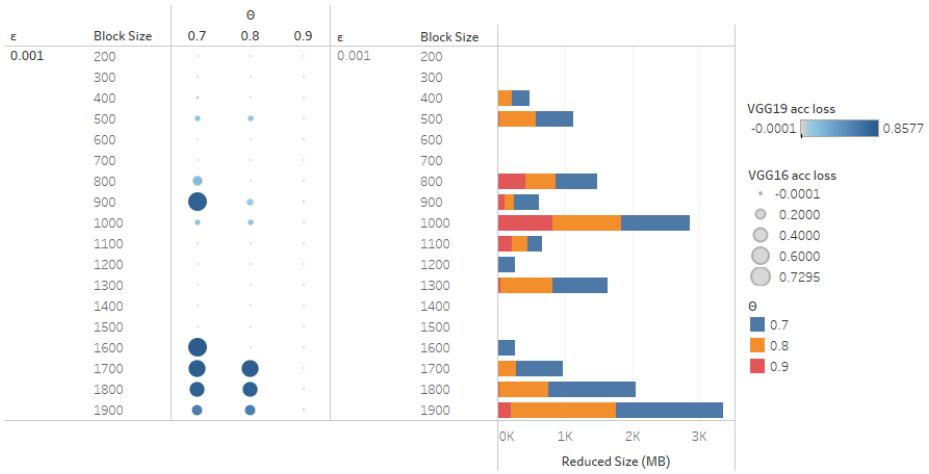


Figure 20: Storage size saved and accuracy loss with various similarity, deduplicating VGG16 and VGG19 models at $\epsilon = 0.001$. $\epsilon = 0.001$ is a decent threshold to deduplicate blocks between VGG16 and VGG19.

square matrices of dimensions 10×10 to 190×190. The deduplication results at each of these dimensions are recorded. Figures 21 and 22 show the total size (MB) saved and accuracy loss of the models at all the similarity percentage thresholds at ϵ thresholds 0.01 and 0.001 respectively. Darker circles show lower accuracy in MNIST Model A. Larger circles denote lower accuracy in MNIST Model B.

Based on these results, both the ϵ thresholds provide approximately the same level of deduplication. Smaller block sizes tend to have better performance of the models after deduplication. With blocking size 20×20, an improvement of $\sim 30\%$ is seen in storage with almost no loss in accuracy.

4.3.6 Conclusion

The blocking granularity must be chosen more intelligently to get the best deduplication performance. The floating-point threshold does not significantly impact the

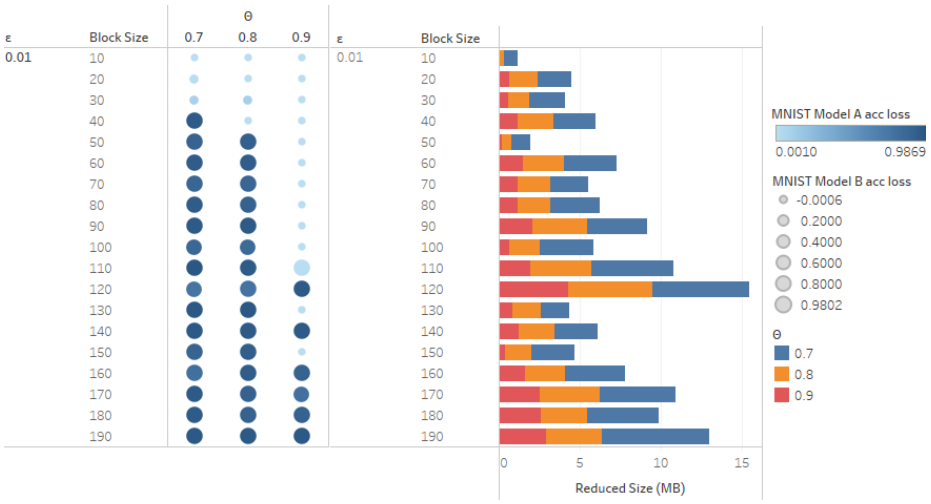


Figure 21: Storage size saved and accuracy loss with various similarity, deduplicating two CNN models at $\epsilon = 0.01$. $\epsilon = 0.01$ is a decent threshold to deduplicate blocks between two same CNN models. Smaller block sizes work better with small models.

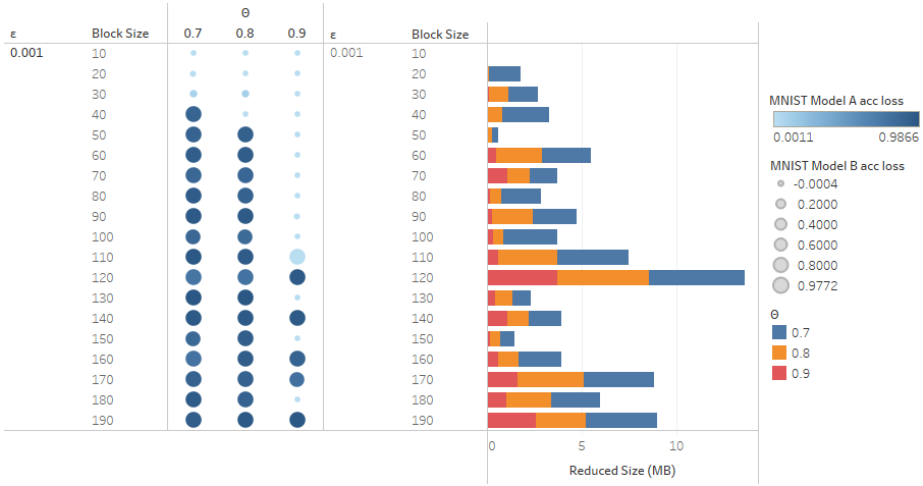


Figure 22: Storage size saved and accuracy loss with various similarity, deduplicating two CNN models models at $\epsilon = 0.001$. $\epsilon = 0.001$ threshold works similar to $\epsilon = 0.01$ threshold.

experiment with two CNN models having the same network, but the results vary significantly in the experiment with VGG16 and VGG19 models.

Chapter 5

SYSTEM DESIGN

5.1 Model Serving on a Database

Model Serving from a Relational Database Management System is not new. Researchers have proposed various ways, from utilizing views provided by the database as models (Deshpande and Madden 2006; Agarwal et al. 2014) to integrating new features such as the ONNX Runtime engine into the database to analyze, optimize and execute the DNN model intermediate representation (Karanasos et al., n.d.). This section describes extending PlinyCompute, a UDF-centric analytics framework using a distributed monolithic storage system called Pangea to implement a Model Serving Platform.

5.1.1 Pangea

Pangea is a distributed monolithic storage system. This system borrows the Query Locality Set Model (QLSM) for buffer management in classical relational database systems. QLSM refers to access patterns and eviction policies for a specific set of pages in a buffer pool. Hence, the locality set describes a set of pages in the buffer pool for a (distributed) file instance. This file instance can be considered a named container that holds the user data or intermediate data distinguished by the container's name, i.e., the container name acts as a namespace that lays claim to some (or all) data. Each set can hold data of a single type defined by the user. This unordered collection of objects

of a specific type can be stored directly into files on the storage medium without any serialization or deserialization (the representation of the object in the memory and the storage is the same). Pangea redefines QLSM for analytical data workloads. Mainly, it allows for large amounts of transient data to be generated without persistence and relaxes the requirement for buffer pool partitioning per locality set (Zou, Iyengar, and Jermaine, n.d.). Pangea also provides a set of services to access the data from a locality set sequentially or shuffle, hash, join and broadcast the data across multiple nodes based on a key. These services allow for pushing computation to the storage layer and handling each data type in a unique way (Barhate 2021).

Popular databases like MySQL and PostgreSQL allow for creating databases that logically group tables into a namespace. Pangea also allows the creation of such named databases. Usually, the word "database" is used to refer to both a database system and a namespace interchangeably. In this thesis, the phrase "named database" refers to a namespace under which multiple locality sets can be created.

5.1.2 PlinyCompute

PlinyCompute is a UDF-centric analytics framework that uses Pangea as its storage system. It implements high-level computations such as Selection, Join, Aggregate, and a few others using the services provided by Pangea. These function similar to the core operations specified by relational algebra (Zou et al. 2018). Using these computations, a user can specify a distributed computation graph of high-level UDFs to operate on the data stored in Pangea. These UDFs are specified using C++ domain-specific lambda calculus. The user-supplied graph of computations is fully optimizable, using many standard techniques from relational query execution.

5.1.3 Lachesis

Lachesis is an extension built on PlinyCompute to address the problem of expensive shuffling operations by automatically partitioning data, which may be complex to map out by hand. It stores the information of the historical executions, its query graph including the file paths, sizes, and runtime statistics such as execution latency and output sizes. Based on this historical information, Lachesis uses a Deep Reinforcement Learning (DRL) approach to optimize the future executions of the same workload by automatically partitioning data horizontally, extracting the keys of the objects to minimize expensive shuffle operations (Zou et al. 2021).

5.1.4 Putting the pieces together

To implement Model Serving from such a system, the system needs to:

1. Be able to represent a DNN model's weight parameters. Since model parameters associated with a layer in a DNN model can be represented as a tensor, user-defined objects that support the representation of tensors can be implemented. These could be stored in Pangea as is. However, with today's state-of-the-art models, a tensor could be huge. The page size used in the buffer pool could be a limitation to store and operate on large tensors in the database. To overcome this issue, the tensor can be split into smaller blocks, each block identified by a unique key representing the position of that tensor block in the bigger tensor. (Yuan et al. 2021) defines this mapping between the key and a tensor block as a tensor relation.

2. Be able to perform the computations defined by the DNN model on the model weight parameters learned in the training process. Machine learning frameworks such

as Tensorflow (older version), Caffe, CNTK, Theano (Paszke et al. 2019) represent the computations defined by the layers in a DNN model as a computation graph.

Common DNN operations (e.g., matrix multiplication, convolution, pooling) operate on tensors. These operations are defined differently in various machine learning frameworks. These are not logical operations; they are physical operations that have to be run as kernel operations. (Yuan et al. 2021) tries to abstract such operations as Tensor Relational Algebra. DNN operations such as matrix multiplication can be modeled as a Join operation and an Aggregation operation in relational algebra. Other DNN operations can be represented as a combination of core relational algebra operators as well (Jankov et al., n.d.).

Since the DNN model parameters, i.e., tensor blocks, can be represented as objects in the database and PlinyCompute accepts a computation graph of relational operations to operate on the objects in a database, any DNN model can be used for inference in this system.

5.2 Extending this system to share Locality Sets

As discussed in 5.1.4, a tensor can be split into tensor blocks. These blocks can be stored in database pages. A collection of these pages that contain tensor blocks of a particular layer's tensor is associated with a distributed file and belongs to a locality set. Pangea can contain multiple such locality sets.

5.2.1 Design of Shareable Pages

When a user loads data associated with multiple locality sets into the database, duplicate data can be grouped to save storage space and improve buffer pool usage. Since user data loaded into Pangea is write-once and read-many, a page containing data shared between multiple locality sets could be loaded into the buffer pool just once and be used in multiple workloads, which reduces the number of disk accesses that would otherwise be required. Page sharing could apply not just to user data but also to the large amounts of transient intermediate data that is generally produced in modern analytics workloads (Zou, Iyengar, and Jermaine, n.d.). On encountering a lack of buffer pool space, transient data would need to be temporarily spilled to disk and loaded back again. Exploiting data overlap to group duplicate data could improve the multi-analytical workload performance.

Sharing of duplicate data can be easily implemented in a traditional RDBMS, where the internal structure of data at the storage level is well defined, and the database itself provides the data types. Deduplication is much harder in UDF-centric databases such as Pangea that allows for storage and computations on user-defined objects. Such user-defined objects are harder to reason with and are opaque to the system. The following few sections introduce a way to deduplicate such user-defined data in Pangea.

5.2.2 The Shared Locality Set

To support sharing of pages across multiple locality sets, the concept of query locality set abstraction is extended to introduce the Shared Locality Set.

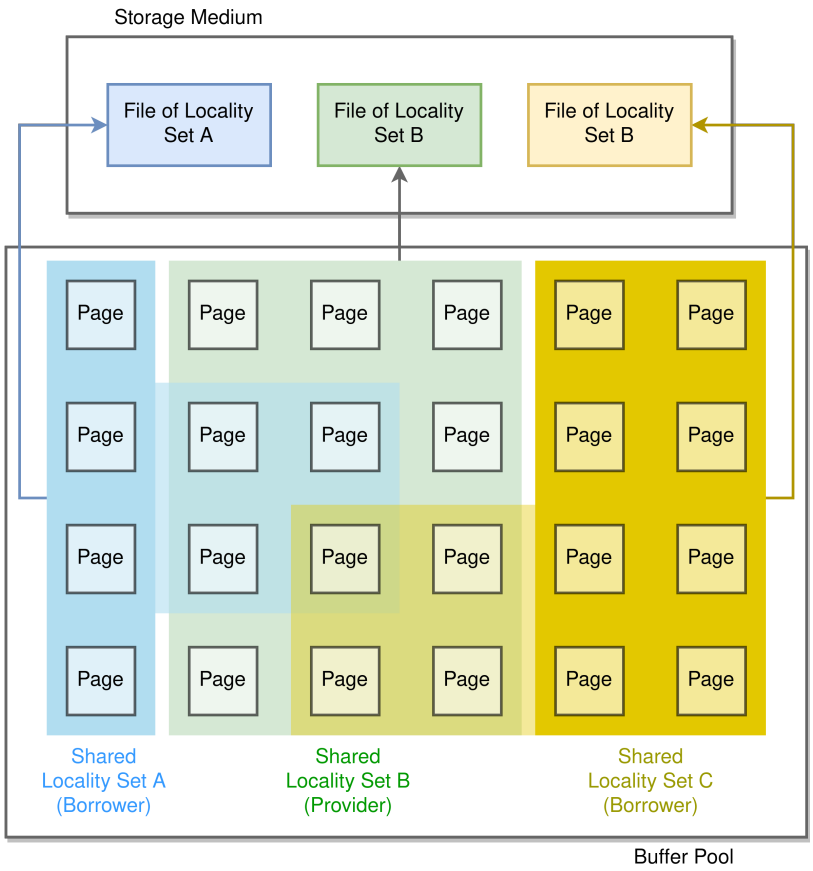


Figure 23: Overlap of Locality Sets in the Buffer pool

A Shared Locality Set supports two roles - a) Borrower - a locality set that borrows pages from one or more locality sets, b) Provider - a locality set that provides pages to one or more locality sets. A Shared Locality Set can play neither, one, or both of these roles at the same time. Hence, a Shared Locality Set may contain pages that belong to itself, borrowed by others, and refers to the pages it borrows from others. A Shared Locality Set also contains added metadata of the pages that it borrows - the actual page identifier (ID) of the page borrowed, the ID of the Provider Shared Locality Set where the borrowed page physically exists, and the ID of the named database that the Provider Shared Locality Set resides.

Extending the locality set to introduce the Shared Locality Set can allow for

different access patterns on its pages. Furthermore, different buffer pool management strategies can be implemented for each access pattern on the Shared Locality Set. For example, depending on the workloads of the different sets playing the borrower role and sharing one locality set, the Shared Locality Set may need to remain in the buffer pool longer than the pages belonging to locality sets that do not play the provider role. Figure 23 shows Shared Locality Sets in the buffer pool.

5.2.3 Shared access to Shared Locality Sets

In Pangea, every workload is modeled as a computation process that has multiple worker threads accessing the buffer pool via shared memory (Zou, Iyengar, and Jermaine, n.d.). Usually, an access control mechanism like locks is used to coordinate read and write access by concurrently running processes or threads to any shared resource. In this case, the shared resource is a buffer pool page shared among multiple locality sets. Many databases, both NoSQL and SQL, like MongoDB ("MongoDB Documentation: Concurrency" 2021), Oracle ("Oracle DB Documentation" 2014) and others solve concurrent read and write operations using a shared-exclusive lock. These locks are a synchronization primitive that allows shared access for read-only use of a resource. Threads that need to write must obtain an exclusive lock on that resource, i.e., only one thread is allowed to make changes to the shared resource. Pangea provides three writing patterns - sequential-write and concurrent-write for write-once immutable data, random-mutable-write for write-many operations, and two reading patterns - sequential-read and random-read. Typical analytical workloads involve a stage of loading an input data set, a sequence of computation stages such as hash aggregations or joins on that input data set, and finally, the result of the

analytical workload is produced. This output can further serve as an input to other analytical workloads. This sequence implies that an input to a stage in an analytical workload is usually stable, i.e., not modified, and the output of every stage serves as an input for the next stage. Hence, to efficiently use the buffer pool space and improve read throughput, pages must be shared after their modification is complete. The workloads utilizing the sequential-write and concurrent-write patterns can be immediately marked as potentially shareable once the page is full. Workloads writing pages with the random-mutable-write pattern must completely release the page before marking the page as shareable.

5.2.4 Shared Locality Set Iterators

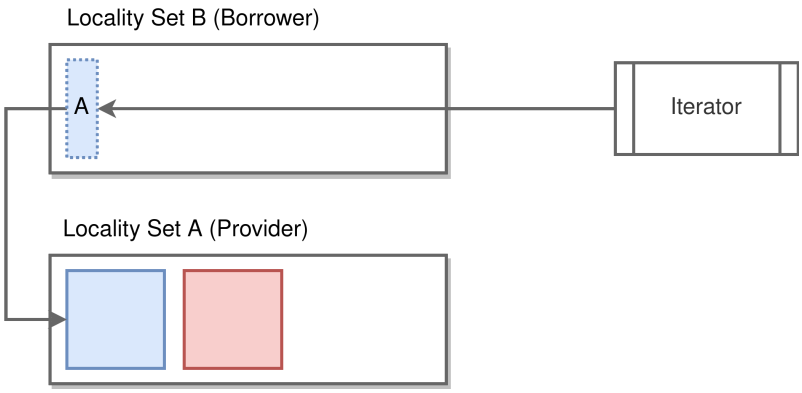


Figure 24: Borrower Locality Set Metadata

The pages of a Locality Set exist in the associated (distributed) file on the disk. These pages may also exist in the buffer pool if the page was recently written into (dirty page) or read (cached page) from the disk for some operation and has not been evicted yet. In addition to these, the pages of a Shared Locality Set may exist on the disk of another file associated with the Provider Shared Locality Set or in the buffer

pool as dirty pages or cached pages due to a recent operation on the Provider Shared Locality Set. To satisfy the two read patterns in Pangea, i.e., sequential-read and random-read patterns, special iterators must fetch the shared pages of the Provider Shared Locality Set from either the disk or the buffer pool. Figure 24 shows how a Shared Locality Set's metadata resolves a page it borrows.

When using user-defined data objects as inputs, computations in Pangea generally use a key to partition data horizontally or perform join or hash-based operations. Select or Project operations in Pangea scan every object to perform its computations. These computations fall under the sequential-read pattern and generally do not require a specific ordering of data to be fed in as inputs. This holds for tensor relational algebra as well, where tensor blocks have a direct relationship with keys that define the positions of these blocks (Yuan et al. 2021). Therefore, the pages in a Shared Locality Set can be accessed in any order for such sequential read patterns. The pages owned by a Shared Locality Set, i.e., the pages stored in a file owned by the set, can be read sequentially with the overhead of one disk seek. However, the borrowed pages from the Provider Locality Sets are a different story. If both the owned pages and borrowed pages reside on the same disk, accessing the borrowed pages could require multiple seeks depending on the number of pages borrowed, the sequence these pages are stored in the corresponding file, and the number of Provider Locality Sets involved. As more and more pages are accessed, transfer time begins to dominate seek time (Abadi et al. 2013). Hence, it is crucial for the performance of the sequential-read pattern on the Shared Locality Set that the borrowed pages stay in the buffer pool for as long as possible. In the case where owned pages and borrowed pages exist on different physical disks, random disk seek latency of the borrowed pages could be hidden under the sequential read of the owned pages, i.e., the borrowed pages can be

buffered while the owned pages are being read from a disk, thereby interleaving the pages read from the Shared Locality Set as a whole.

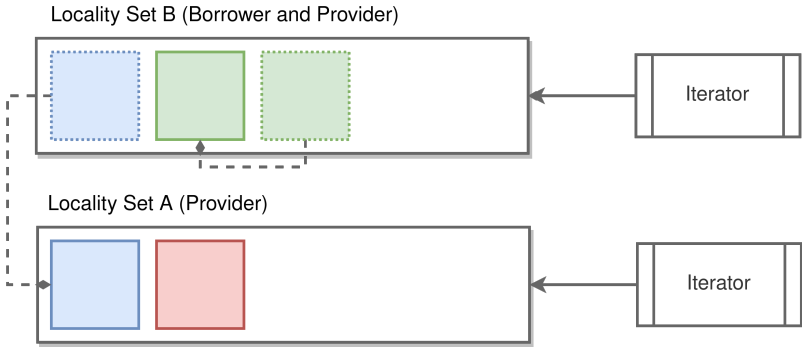


Figure 25: Virtualization of Locality Sets

All the pages of a Locality Set have a unique page ID of their own. Shared Locality Sets borrow pages from other sets. Borrowing pages pose a problem when an operation in Pangea needs random access to specific pages—for example, spilling intermediate data to disk due to lack of space in the buffer pool and reading that data again to resume the operation. The operation needs a way to read a specific page in the Shareable Locality Set. The operation that spilled the intermediate data thinks that it wrote a page with a specific page ID, but that page may not have been written to disk and discarded because it was detected as a duplicate page of an already existing page in the system. The page ID that the discarded page had can be used as a "virtual" page ID pointing to the already existing page to deal with this scenario. This provides a form of virtualization, where the Shareable Locality Set seems to encompass all the pages written to it, but underneath, the virtual page ID is a pointer to another page ID of the Provider Shared Locality Set. Figure 25 shows the virtualized view of a Shared Locality Set to someone who uses it. The dotted blocks do not exist, but any

operation that wants to access a specific page in a Locality Set can use page IDs that may be real or virtual.

5.2.5 Detecting duplicate pages

In Pangea, a user loads a collection of user-defined objects into a named database through a client. There is no concept of a database page here just yet. The user specifies a name of the Locality Set that these objects must reside in for later use. The client sends this collection of objects to a dispatcher that decides on the partitioning of the objects (Zou et al. 2021). Once the objects are partitioned, they are ready to be sent to the storage servers. So if there exists multiple servers, a subset of the collection of objects sent by the user is available on one specific server. On one storage server, these objects are buffered in memory for a lazy flush. Once there is no more space in the buffer, these objects are written onto a page.

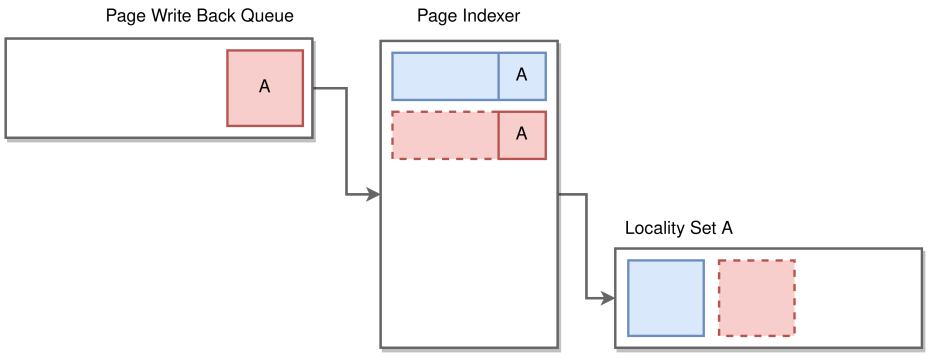


Figure 26: Keeping track of unique page content

The database needs the ability to uniquely identify these pages based on their content to deduplicate them. This unique identification will help the database decide if a page is to be flushed to the disk or not. However, the database cannot reason with

user-defined objects independently, as it does not know its structure. These objects can be arbitrary, and the understanding of the object's composition is necessary to identify a collection of such objects uniquely. Solving this problem requires intervention from the user.

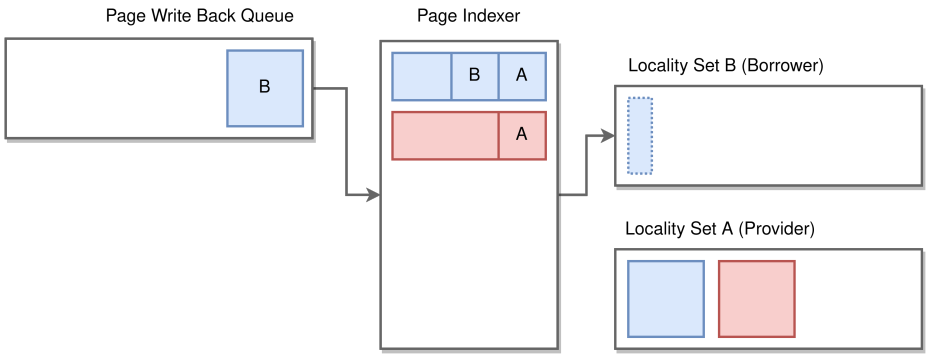


Figure 27: Eliminating duplicate pages

A `PageIndexer` interface (defined in 5.1) is used internally by the database to maintain a mapping between a unique page content identifier (PCID) and a list of all the page IDs that have the same content. A `PageIndexer` instance is associated with a named database and user-defined type pair. This means a `PageIndexer` instance will manage deduplication of all Locality Sets under one named database, containing a specific user-defined type. The first occurrence of a page with a specific PCID is physically present on the disk as shown in Figure 26. The rest are virtual page IDs as discussed in 5.2.4. So, the page's content can be hashed in some way to generate a PCID when that page is ready to be flushed. This page can be discarded if the same PCID present in the `PageIndexer`. This operation is shown in Figure 27.

The system uses this interface to add and query pages while pages arrive to be written into the Shared Locality Sets. A way to identify a collection of user-defined objects uniquely is required to eliminate duplicate pages. Since a database cannot

Listing 5.1: The PageIndexer Interface

```
1  template <class KeyType>
2  class PageIndexer {
3    std::unordered_map<KeyType, std::vector<SharedPageID>> *pageMap;
4  public:
5    PageIndexer() {}
6    ~PageIndexer() {}
7
8    virtual void initialize() = 0;
9    virtual void addPage(PDBPagePtr page) = 0;
10   virtual SharedPageID* checkAndAddPage(PDBPagePtr page) = 0;
11   virtual SharedPageID* queryPage(PDBPagePtr page) = 0;
12 };
```

make assumptions about the user-defined objects stored in these pages, an option is provided to generate a hash of a user-defined object or a collection of user-defined objects that reside on a page. To this end, the `PageIndexer` interface (defined in 5.2) is further extended to contain two more methods - `hashObject` and `hashObjects`. The implementation of these methods is the responsibility of the user.

The extension of the `PageIndexer` to include both the option to generate hashes for one user-defined object and multiple user-defined objects opens up several approaches to generate the PCID of a database page. If the user implements `hashObjects` for the user-defined objects, the system can use this to generate page signatures. If the user implements just the `hashObject` method, the system can fall back to other resemblance detection techniques. These techniques could be used to enforce exact page matching or relaxed to re-group user-defined objects into newer pages if the overall storage used in the database could be reduced.

A basic technique to detect duplicate pages could be to maintain a mapping of the object hashes to the list of its Page IDs. Incoming objects can be hashed, and an intersection of the Page ID list associated with these hashes can be used to find

Listing 5.2: The Extended PageIndexer Interface

```
template <class KeyType, class ValueType>
class ExtendedPageIndexer : public PageIndexer {
public:
  ExtendedPageIndexer() {}
  ~ExtendedPageIndexer() {}

  virtual KeyType* hashObject(Handle<ValueType> &objects) = 0;
  virtual KeyType* hashObjects(Vector<Handle<ValueType>> &objects) = 0;
};
```

duplicate or similar pages. The strictness of this technique could be adjusted to reshuffle more common objects into newer pages.

Another technique could be to use a super feature approach. The hashes of all the objects in a page could be deterministically sampled and coalesced into a super feature, also referred to as super fingerprint, and then indexing the super features to detect duplicate or highly similar pages (Xia et al. 2016). An example of a sampling technique to find a representative super fingerprint could be Broder's theorem. Broder's theorem states - the probability that two sets S_1 and S_2 have the same minimum hash element is the same as their Jaccard similarity coefficient (Bhagwat et al. 2009).

$$Pr[min(hash(S_1) = min(hash(S_2)] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{5.1}$$

So, if two pages with user-defined objects have the same minimum hash, then the two pages are highly similar. Then the similar objects could be grouped to a newer page if the overall storage space used can be reduced. For exact match deduplication, once a possible page match is found using Broder's theorem, the hashes of each of the objects in the two pages must be compared to confirm deduplication eligibility.

5.2.6 Fluid roles of a Shared Locality Set

The role of a Shared Locality Set is not fixed. As mentioned in 5.2.2, a Shared Locality Set can either be a provider, a borrower, both, or neither. As new sets are written into, and existing sets are deleted from the database, the role of a Shared Locality Set can change.

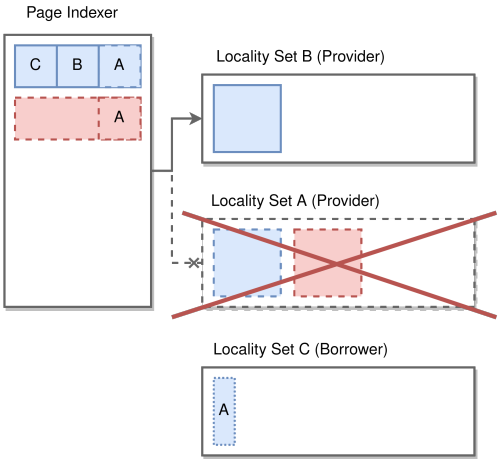


Figure 28: Example of a role change

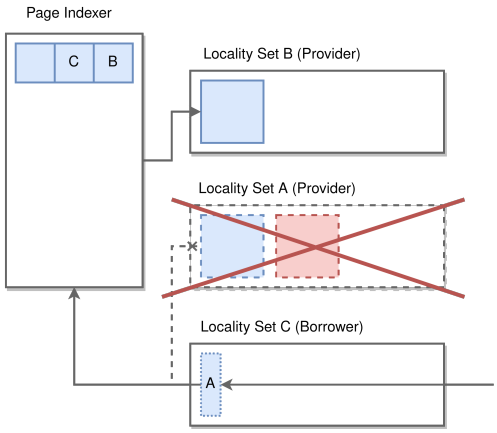


Figure 29: Example of a Page fault

When a new page is ready to be flushed to disk and detected as a duplicate of an already existing page on disk (or cache), the Locality set that owns this duplicate page becomes a borrowing set. Then, the Locality set that owns the already existing page on the disk (or cache) becomes a provider set. This sequence is shown in Figure 27.

When a Provider Shared Locality Set is deleted, the pages in use by the Borrower Shared Locality Sets must remain in the database. Each of these pages is moved to the Borrower Locality Sets that first borrowed this page. Thus, these Borrower Locality Sets now play the role of a Provider as shown in Figure 28. The `PageIndexer` keeps a record of all the pages being borrowed from this Shared Locality Set. When

a Borrower Locality set tries to lookup a page of a Locality Set that was recently deleted, a "page fault" event is triggered as shown in Figure 29. The Borrower Shared Locality Set then looks to the `PageIndexer` to resolve the Shared Locality Set where the page is located, and all the other entries corresponding to the deleted Shared Locality Set are updated to reflect the Shared Locality Set that now plays the role of a provider.

Chapter 6

SYSTEM EVALUATION AND DISCUSSION

6.1 Environment Setup

Experiments are run on AWS with two identical r4.2xlarge EC2 instances. One of them acts as a primary (`pdb-cluster`), which accepts computations specified by a client and coordinates them among the secondary servers. The other EC2 instance acts as a secondary (`pdb-server`), which stores all the data provided by the client and runs the computations on the stored data as specified by the primary. The client program is also run on the EC2 instance that runs the primary.

Both the EC2 instances have 8 vCPU cores and 61 GB RAM each. The storage volumes attached to both these EC2 instances are unencrypted and of type gp2 (SSD) with 384 IOPS, 128 GB capacity. These EC2 instances are connected with 10 Gbps Ethernet links. The master branch of Lachesis was used as a base for the system design specified in chapter 5.

6.2 Experiment Description

Similar to (Yuan et al. 2021), a simple two-layer feed-forward neural network (FFNN) is implemented in Lachesis for multiple label classification. This FFNN requires five matrices, the two weights and biases of the two layers, and the input for which predictions are generated. Suppose the input matrix includes N data points having F features, then the input matrix $X \in \mathbb{R}^{N \times F}$. If there are L labels, the

weight matrices are $W_1 \in \mathbb{R}^{F \times H}$ and $W_2 \in \mathbb{R}^{H \times L}$ where H is the hidden layer size. The biases are $B_1 \in \mathbb{R}^{H \times 1}$ and $B_2 \in \mathbb{R}^{L \times 1}$. This results in an output matrix that is $Y \in \mathbb{R}^{N \times L}$ as the one-hot encoded labels for the inputs.

The sequence of TRA computations required for the inference of the FFNN is (Yuan et al. 2021):

$$R_{a_1} = \lambda_{(relu)} \left(\sum_{(\langle 0,\langle 2,1 \rangle \rangle, matAdd)} \left(\sum_{(\langle 0,2 \rangle, matAdd)} \left(\bowtie_{(\langle 1 \rangle, \langle 0 \rangle)), matMul} \right) (R_X, R_{W_1}) \right), R_{B_1} \right)$$

$$R_Y = \lambda_{(sigmoid)} \left(\sum_{(\langle 0,\langle 2,1 \rangle \rangle, matAdd)} \left(\sum_{(\langle 0,2 \rangle, matAdd)} \left(\bowtie_{(\langle 1 \rangle, \langle 0 \rangle)), matMul} \right) (R_{a_1}, R_{W_2}) \right), R_{B_2} \right)$$

The models used in these experiments have feature size $F = 597,540$, label size $L = 14,588$ with hidden layer size $H = 1000$. The batch size used is $N = 1000$.

A Transfer Learning Scenario is tested, where W_1 is shared, and W_2 is specialized for different tasks. The inputs, weights, and biases are randomly generated using a `uniform_real_distibution` generator in C++ with the range -0.5 and 0.5 as the goal of this experiment is to establish the performance of this system. There is no effect on accuracy in a transfer learning scenario where the pages of W_1 are shared completely because W_1 used in all the models are the same.

In these experiments, storage buffer pool sizes used are 8192 MB, 9216 MB, 10240 MB, 11264 MB, 12288 MB, 13312 MB, and 14336 MB. The thread counts used are 4 and 8 threads. Experiments are also run with two, three, and four specialized FFNN sharing a single W_1. All combinations of these parameters are used to gather the results of these experiments.

A Model Serving Workload is executed where the inference on all the models is invoked twice. For example, if there were three models in the database, M_1, M_2, and M_3 are invoked in sequence. Then this sequence is repeated once more.

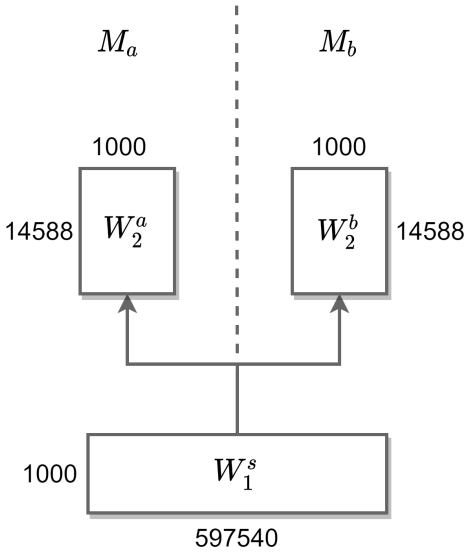


Figure 30: Example of a two model Transfer Learning scenario used in the experiments

To evaluate the runtime performance of the Model Serving Workload, exact matching of pages is implemented through the `PageIndexer` interface. This works well for a transfer learning scenario since whole layers are shared.

6.3 Results

6.3.1 Cache Hit improvements

Based on the results in Figure 31, cache hits massively improve with sharing pages, especially if many models share a set of pages. With four models sharing W_1 and buffer pool size 14GB, the system manages 316 page cache hits, out of which 216 were shared pages, and 524 page cache misses. The no page sharing scenario of the same workload and buffer pool size achieves just 128 page cache hits with 752 page cache misses. Page hits improve by 146% in this workload.

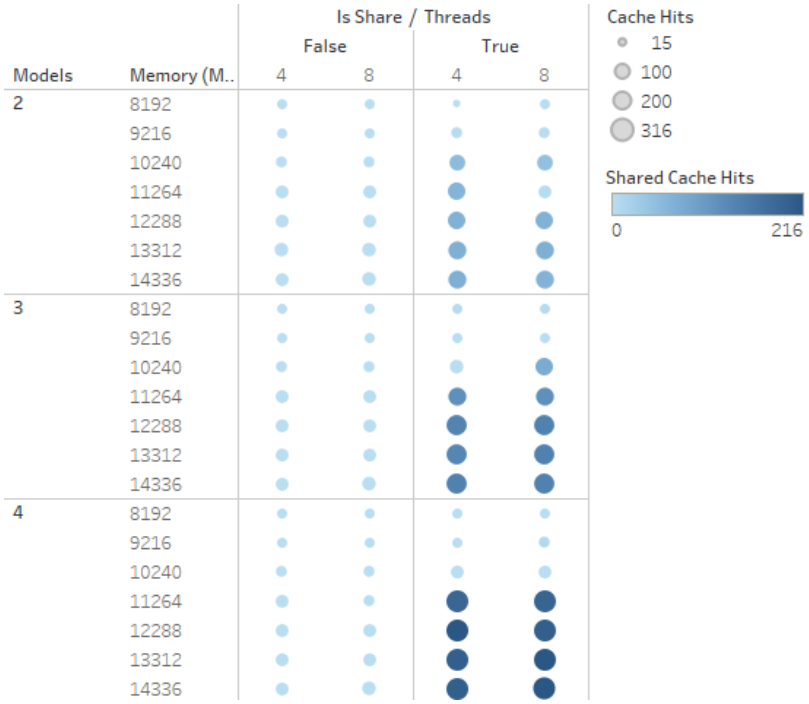


Figure 31: Page Cache hits in the shared pages and no shared pages scenario. Larger the circle, the higher the overall cache hits. Darker the circle, the higher the shared page cache hits. There are higher cache hits with a large enough buffer pool size.

6.3.2 Performance improvements

As seen in Figure 32, there is a consistent improvement in the runtime performance of reusing pages (and consequently the shared matrix blocks in memory) for repeated computations. With buffer pool sizes 8192 MB and 9216 MB, improvement is not noticeable due to page sharing because shared pages are evicted from memory to make room for locality sets of weights that are needed in the computation of every layer and pages required to hold the generated intermediate outputs. However, with larger buffer pool sizes, the benefits of sharing pages are apparent. With a buffer pool size of 13312 MB, model inference time reduces from 109 seconds to $\sim$ 92 seconds - a 15%

gain in runtime on an SSD with 384 IOPS. A more significant gain can be expected on a magnetic hard disk because page read times are much higher on such disks.

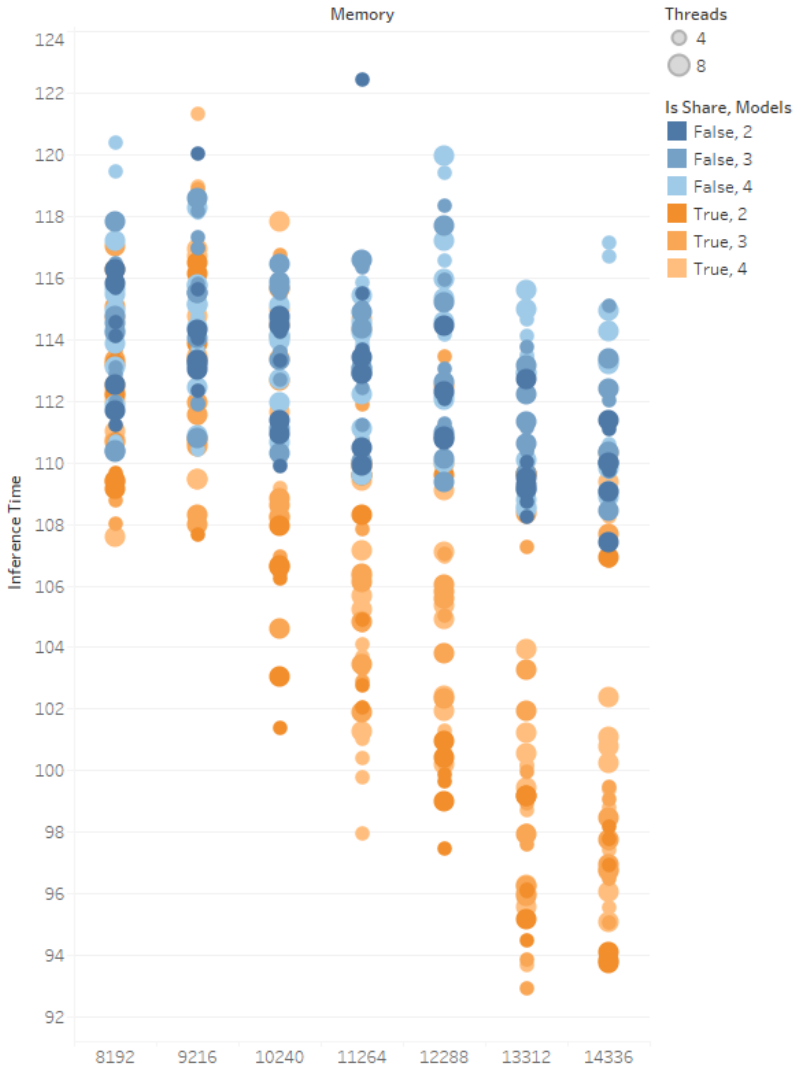


Figure 32: Workload Inference time in the shared pages and no shared pages scenario. Blue shades show runtime for the various number of models used in the workload in a no-sharing scenario. Orange shades show runtime for the various number of models used in the workload in a sharing scenario. The circle size denotes the number of threads used for that workload. Corresponding to the number of cache hits in Figure 31, larger improvements in runtime are observed.

6.3.3 Overheads

6.3.3.1 Offline overhead - The PageIndexer

The `PageIndexer` contains a mapping between a hash of a page and the relevant information of the sets that require this page. The `PageIndexer` is implemented using C++ STL `std::unordered_map`. The set information stored in this implementation is the database ID, set ID, and page ID. For exact page matching, the key is just a hash of the page generated as a `long` type which is of 8 bytes. Each of the identifiers (database, set, and page) are of type `unsigned int`, which is of 4 bytes each. Hence, every shareable page takes up a space of $8 + 12 = 16$ bytes. Additional 12 bytes is used for every set that shares a page.

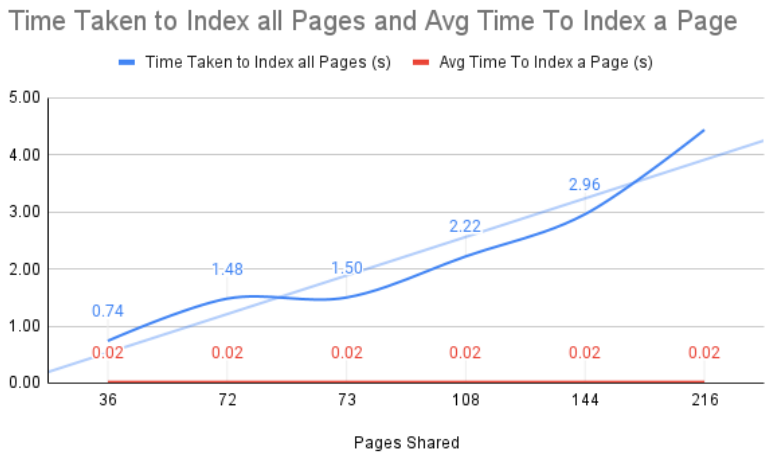


Figure 33: Overhead of Building the PageIndexer - There is a linear increase in time to build the PageIndexer as the number of pages increases.

Every insert to the `std::unordered_map` is amortized constant on average, worst case linear in the size of the container ("C++ Reference" 2021). This index could

get harder to maintain if many pages are not shared, due to the creation of an entry for every page in the database. However, this overhead is encountered offline while loading the model into the database and not during inference.

6.3.3.2 Online overhead - The Borrower Shareable Set Metadata

Every Borrower Shareable Set maintains metadata that references all the pages borrowed from other Shareable Sets. This is implemented using C++ STL `std::vector` of shared page information. Just like in 6.3.3.1, this metadata stores a list of relevant information of the sets from which the pages are borrowed. This information includes the database ID, set ID, and page ID i.e. a total of 12 bytes for every page borrowed.

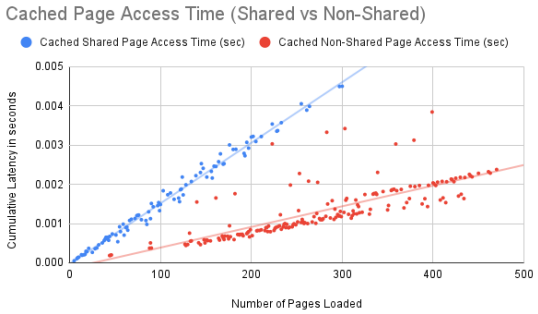


Figure 34: Cached Page Access Time - Shared vs Non-Shared: Overhead exists in accessing cached pages due to indirection.

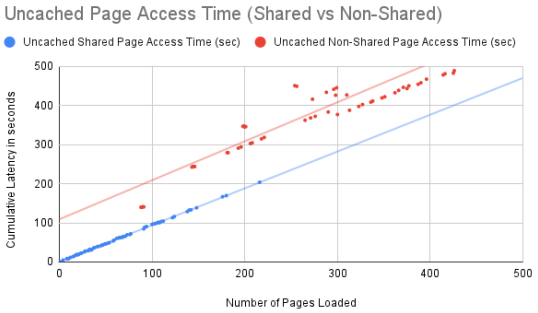


Figure 35: Uncached Page Access Time: Shared vs Non-Shared - Improvement in cache miss of shared pages most likely due to the internal caching mechanisms used by an SSD while reusing a page.

A `std::vector` has an amortized insert time complexity due to the need to resize the container to fit in new elements ("C++ Reference" 2021). Iterating over each shared page information has a constant lookup time ("C++ Reference" 2021) and

adds a small overhead of $\sim 1.02 \times 10^{-5}$ seconds per page accessed. Still, about 300 cached shared pages can be loaded by the system in a total time of ~ 0.004 seconds.

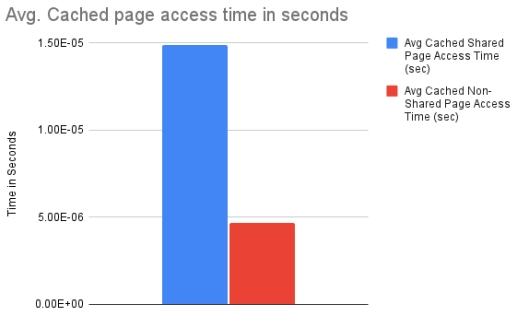


Figure 36: Single Cached Page Access Time: Shared vs Non-Shared - A neglectable overhead due to indirection in accessing the metadata and the required file.

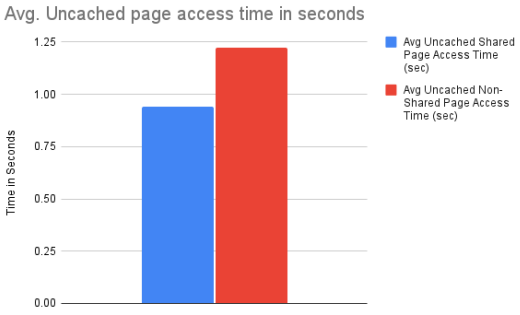


Figure 37: Single Uncached Page Access Time: Shared vs Non-Shared - Improvement in cache miss of shared pages most likely due to the internal caching mechanisms used by an SSD while reusing a page.

There is an improvement to cache miss and time to load a shared page as compared to an unshared page from an SSD. An SSD may likely cache frequently or most recently accessed data to serve the cache miss faster in the case of shared pages.

Chapter 7

CONCLUSION AND FUTURE SCOPE

This thesis shows the possibility of similar corresponding layers in DNN models of similar architectures operating on the same or different datasets. Layer similarity is also possible when incremental changes are applied to DNN models by training an existing model on new data to create a newer version of the DNN model. The page sharing experiments also show that, even if the DNN models have the same architecture, the layer similarity depends on the hyperparameters used for training these models.

Inter-model and intra-model similarity can be exploited by blocking the model weight parameters into tensor relations. Based on the experiments of merging two CNN models with the same architecture and merging a VGG16 model and a VGG19 model, unique tensor blocks can be used to replace similar blocks without much loss in accuracy while significantly reducing storage requirements. However, the tolerance to block deduplication varies by model. Perhaps in the future, experiments can be conducted to verify the drop in accuracy due to block deduplication by measuring the importance (e.g., fisher information as used in (Lee and Nirjon 2020)) of the blocks being replaced.

An extension to the Query Locality Set Model (QLSM) is also presented to exploit the similarity between the layers of DNN models to reduce storage and improve model serving performance by sharing database pages. The proposed design of the extension applies not just to DNN models but to any kind of data being stored in a database. This extension seamlessly integrates with the current design of Pangea, a monolithic

distributed storage system, with minimal changes to the existing architecture to deduplicate pages as they arrive to be stored automatically.

7.1 Future Work

The proposed extension currently deduplicates pages in the order they arrive in the storage system. This process can be further improved to increase the potency of deduplication in a database management system. Thus, the problem deduplication of any database objects —any user-defined objects or inbuilt data types provided by a database —can be defined at two levels of storage.

7.1.1 Deduplication at local storage

Given many sets of objects, each set associated with a Locality Set, a database needs to store these objects into a set of pages. In the typical case, mutually exclusive subsets of these pages are associated with different Locality Sets. If duplicate objects exist within and across sets of objects, is it possible to reorganize the placement of each object, eliminating duplicates, into pages such that overlapping subsets of pages define different Locality Sets? This question could be modeled as a bin-packing problem (Sindelar, Sitaraman, and Shenoy 2011) where objects are packed into pages such that the number of pages is minimized, keeping in mind that the objects placed in a page could have size $<$ sum of their individual sizes due to deduplication. Hence, this is an NP-Hard problem.

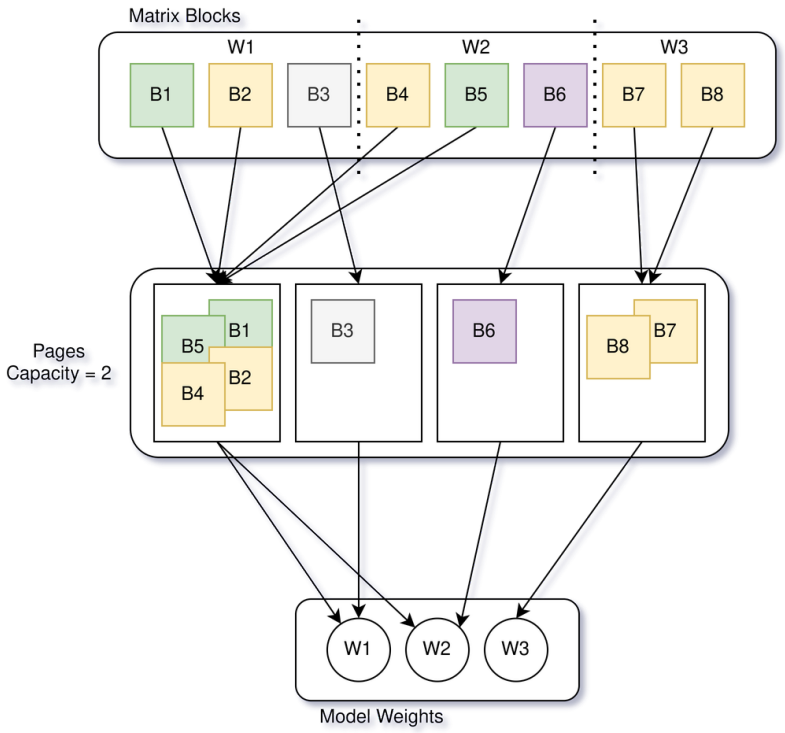


Figure 38: Local Deduplication problem

7.1.2 Deduplication at distributed storage

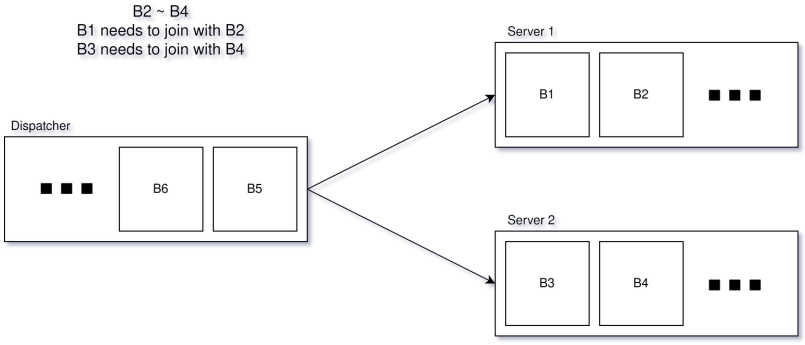


Figure 39: Distributed Deduplication problem

Lachesis focuses on effectively partitioning database objects to ensure minimal

shuffling overhead on workloads with computations like joins or aggregations. Deduplication of database objects, as defined in 7.1.1, tries to pack objects in storage such that a minimal number of pages are used to represent the sets of objects stored by a user logically. These may be conflicting goals. For example, in Figure 39, object B1 needs to join with object B2 and object B3 needs to join with object B4. Objects B2 and B4 are duplicates. Lachesis decides the partitioning keeping in mind the efficiency of the entire workload, and so places B1 and B2 in server 1 and B3 and B4 in server 2. However, deduplication cannot occur with this partitioning. Balancing the two components —partitioning to reduce shuffle overhead and deduplication to reduce storage overhead —is another open question for further research.

REFERENCES

Abadi, D, PA Boncz, S Idreos, S Harizopoulos, and S Madden. 2013. "The Design and Implementation of Modern Column-Oriented Database Systems." *Foundations and Trends in Databases* 5 (3): 197–280.

Abadi, Martın, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467.*

Agarwal, Deepak, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. 2014. "Laser: A scalable response prediction platform for online advertising," 173–182.

Alam, Aftab, Shah Khalid, Muhammad Numan Khan, Tariq Habib Afridi, Irfan Ullah, and Young-Koo Lee. 2020. "Video Big Data Analytics in the Cloud: Research Issues and Challenges." *arXiv e-prints,* arXiv–2011.

Allen-Zhu, Zeyuan, and Yuanzhi Li. 2020. "Towards Understanding Ensemble, Knowledge Distillation and Self-Distillation in Deep Learning." *arXiv preprint arXiv:2012.09816.*

Arai, Kohei, and Supriya Kapoor. 2019. "Advances in Computer Vision," 15.

"AWS Documentation: Training ML Models." 2021, June. https://docs.aws.amazon.com/machine-learning/latest/dg/training-ml-models.html.

"AWS SageMaker." 2021, June. https://aws.amazon.com/sagemaker/.

Barhate, Pratik N. 2021. "Shuffle Overhead Analysis for the Layered Data Abstractions." *ProQuest Dissertations and Theses,* 53. http://login.ezproxy1.lib.asu.edu/login?url=https://www-proquest-com.ezproxy1.lib.asu.edu/dissertations-theses/shuffle-overhead-analysis-layered-data/docview/2533323197/se-2?accountid=4485.

Bhagwat, Deepavali, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. 2009. "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," 1–9.

Bianco, Simone, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. "Benchmark analysis of representative deep neural network architectures." *IEEE Access* 6:64270–64277.

Borthwick, Andrew, Stephen Ash, Bin Pang, Shehzad Qureshi, and Timothy Jones. 2020. "Scalable Blocking for Very Large Databases." *arXiv preprint arXiv:2008.08285.*

"C++ Reference." 2021, July. https://en.cppreference.com/w/cpp/container/unordered_map/emplace.

"C++ Reference." 2021, July. https://en.cppreference.com/w/cpp/container/vector/push_back.

"C++ Reference." 2021, July. https://en.cppreference.com/w/cpp/container/vector/operator_at.

Canziani, Alfredo, Adam Paszke, and Eugenio Culurciello. 2017. "An Analysis of Deep Neural Network Models for Practical Applications," arXiv: 1605.07678 [cs.CV].

Chaalal, Hichem, Mostefa Hamdani, and Hafida Belbachir. 2020. "Finding the best between the column store and row store Databases," 1–4.

Chard, Ryan, Zhuozhao Li, Kyle Chard, Logan Ward, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J Franklin, and Ian Foster. 2019. "DLHub: Model and data serving for science," 283–292.

Cheng, Yu, Duo Wang, Pan Zhou, and Tao Zhang. 2017. "A survey of model compression and acceleration for deep neural networks." *arXiv preprint arXiv:1710.09282.*

Choi, Yoojin, Mostafa El-Khamy, and Jungwon Lee. 2020. "Learning sparse low-precision neural networks with learnable regularization." *IEEE Access* 8:96963–96974.

Crankshaw, Daniel, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. "Clipper: A low-latency online prediction serving system," 613–627.

"Deep learning." 2021, June. https://en.wikipedia.org/wiki/Deep_learning#History.

Deshpande, Amol, and Samuel Madden. 2006. "MauveDB: supporting model-based user views in database systems," 73–84.

Do, Jaeyoung, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. 2011. "Turbocharging DBMS buffer pool using SSDs," 1113–1124.

Fan, Jianqing, Cong Ma, and Yiqiao Zhong. 2019. "A Selective Overview of Deep Learning." *stat* 1050:15.

Fang, Jiarui, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. "TurboTransformers: an efficient GPU serving system for transformer models," 389–402.

"Floating-Point Formats and Deep Learning." 2020, July. https://eigenfoo.xyz/floating-point-deep-learning/.

Ganesh, Prakhar, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Preslav Nakov. 2020. "Compressing large-scale transformer-based models: A case study on bert." *arXiv preprint arXiv:2002.11985.*

"Google Protobuf Documentation: Encoding." 2021, June. https://developers.google.com/protocol-buffers/docs/encoding.

"Google Protobuf Repository." 2021, June. https://github.com/protocolbuffers/protobuf.

"GPU Shortage." 2020, October. https://www.theverge.com/2020/10/5/21503001/nvidia-rtx-3080-and-3090-supply-issues-2021.

Guliyev, Namig J, and Vugar E Ismailov. 2016. "A single hidden layer feedforward network with only one neuron in the hidden layer can approximate any univariate function." *Neural computation* 28 (7): 1289–1304.

Guo, Kaiyuan, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. "A survey of FPGA-based neural network accelerator." *arXiv preprint arXiv:1712.08934.*

Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. "Deep learning with limited numerical precision," 1737–1746.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. "Deep residual learning for image recognition," 770–778.

He, Yihui, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. "Amc: Automl for model compression and acceleration on mobile devices," 784–800.

Ishakian, Vatche, Vinod Muthusamy, and Aleksander Slominski. 2018. "Serving Deep Learning Models in a Serverless Platform," 257–262.

Jankov, Dimitrije, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2020. "Declarative Recursive Computation on an RDBMS:

or, Why You Should Use a Database For Distributed Machine Learning." *ACM SIGMOD Record* 49 (1): 43–50.

Jankov, Dimitrije, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. n.d. "Declarative Recursive Computation on an RDBMS." *Proceedings of the VLDB Endowment* 12 (7).

Karanasos, Konstantinos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. n.d. "Extending Relational Query Processing with ML Inference." *Optimization* 2 (4): 7.

"Keras documentation: Transfer learning & fine-tuning." 2021, June. https://keras.io/guides/transfer_learning/.

Khan, Awais, Chang-Gyu Lee, Prince Hamandawana, Sungyong Park, and Youngjae Kim. 2018. "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," 87–93.

Kim, Mijung. 2014. "TensorDB and Tensor-Relational Model (TRM) for Efficient Tensor-Relational Operations."

Klabjan, Diego, and Xiaofeng Zhu. 2020. "Neural Network Retraining for Model Serving." *arXiv preprint arXiv:2004.14203.*

Kolb, Lars, Andreas Thor, and Erhard Rahm. 2012. "Dedoop: Efficient deduplication with hadoop." *Proceedings of the VLDB Endowment* 5 (12): 1878–1881.

Koo, Dongyoung, and Junbeom Hur. 2018. "Privacy-preserving deduplication of encrypted data with dynamic ownership management in fog computing." *Future Generation Computer Systems* 78:739–752. https://doi.org/https://doi.org/10.1016/j.future.2017.01.024.

Krishnan, Sanjay, and Eugene Wu. 2019. "Alphaclean: Automatic generation of data cleaning pipelines." *arXiv preprint arXiv:1904.11827.*

Lee, Seulki, and Shahriar Nirjon. 2020. "Fast and scalable in-memory deep multitask learning via neural weight virtualization," 175–190.

LeMay, Matthew, Shijian Li, and Tian Guo. 2020. "PERSEUS: Characterizing Performance and Cost of Multi-Tenant Serving for CNN Models."

Li, Peng, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. "CleanML: a study for evaluating the impact of data cleaning on ml classification tasks."

Loshchilov, Ilya, and Frank Hutter. 2017. "Decoupled weight decay regularization." *arXiv preprint arXiv:1711.05101.*

Lu, Jie, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. "Learning under concept drift: A review." *IEEE Transactions on Knowledge and Data Engineering* 31 (12): 2346–2363.

Miao, Hui, Ang Li, Larry S Davis, and Amol Deshpande. 2017. "Towards Unified Data and Lifecycle Management for Deep Learning," 571–582.

Mishra, Rahul, Hari Prabhat Gupta, and Tanima Dutta. 2020. "A Survey on Deep Neural Network Compression: Challenges, Overview, and Solutions." *arXiv preprint arXiv:2010.03954.*

"MongoDB Documentation: Concurrency." 2021, June. https://docs.mongodb.com/manual/faq/concurrency/.

"Moore's law." 2021, May. https://en.wikipedia.org/wiki/Moore's_law.

Neill, James O'. 2020. "An overview of neural network compression." *arXiv preprint arXiv:2006.03669.*

"ONNX." 2021, May. https://onnx.ai/about.html.

"Oracle DB Documentation." 2014, June. https://docs.oracle.com/database/121/TTCIN/concurrent.htm#TTCIN176.

Padhy, Rabi Prasad, Manas Ranjan Patra, and Suresh Chandra Satapathy. 2011. "RDBMS to NoSQL: reviewing some next-generation non-relational database's." *International Journal of Advanced Engineering Science and Technologies* 11 (1): 15–30.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32:8026–8037.

"Python Documentation: pickle." 2021, June. https://docs.python.org/3/library/pickle.html.

"PyTorch Documentation: Saving and Loading Models." 2021, June. https://pytorch.org/tutorials/beginner/saving_loading_models.html.

Qararyah, Fareed, Mohamed Wahib, Doğa Dikbayır, Mehmet Esat Belviranli, and Didem Unat. 2021. "A computational-graph partitioning method for training memory-constrained DNNs." *Parallel Computing* 104:102792.

Ramachandra, Karthik, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. "Froid: Optimization of imperative programs in a relational database." *Proceedings of the VLDB Endowment* 11 (4): 432–444.

"Rethink Data: Seagate US." 2020, July. https://www.seagate.com/files/www-content/our-story/rethink-data/files/Rethink_Data_Report_2020.pdf.

Rhu, Minsoo, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," 1–13.

Romero, Francisco, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2020. "INFaaS: A Model-less and Managed Inference Serving System," arXiv: 1905.13348 [cs.DC].

Schatz, Martin D, Robert A Van de Geijn, and Jack Poulson. 2016. "Parallel matrix multiplication: A systematic journey." *SIAM Journal on Scientific Computing* 38 (6): C748–C781.

Shen, Haichen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis," 322–337.

Shin, Youngjoo, Dongyoung Koo, and Junbeom Hur. 2017. "A survey of secure data deduplication schemes for cloud storage systems." *ACM computing surveys (CSUR)* 49 (4): 1–38.

Simeone, Osvaldo. 2018a. "A Brief Introduction to Machine Learning for Engineers."

———. 2018b. "A Very Brief Introduction to Machine Learning With Applications to Communication Systems." *IEEE Transactions on Cognitive Communications and Networking* 4 (4).

Sindelar, Michael, Ramesh K Sitaraman, and Prashant Shenoy. 2011. "Sharing-aware algorithms for virtual machine colocation," 367–378.

Sze, Vivienne, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. "Efficient processing of deep neural networks: A tutorial and survey." *Proceedings of the IEEE* 105 (12): 2295–2329.

Tae, Ki Hyun, Yuji Roh, Young Hun Oh, Hyunsu Kim, and Steven Euijong Whang. 2019. "Data cleaning for accurate, fair, and robust models: A big data-AI integration approach," 5–1.

Tambe, Thierry, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2019. "Adaptivfloat: A floating-point based data type for resilient deep learning inference." *arXiv preprint arXiv:1909.13271.*

"TensorFlow Documentation: Asset." 2021, June. https://www.tensorflow.org/api_docs/python/tf/saved_model/Asset.

"TensorFlow Documentation: Checkpoints." 2021, June. https://www.tensorflow.org/guide/checkpoint#saving_from_tfkeras_training_apis.

"TensorFlow Documentation: SavedModel." 2021, June. https://www.tensorflow.org/guide/saved_model#the_savedmodel_format_on_disk.

"TensorFlow Documentation: Saving Models." 2021, June. https://www.tensorflow.org/guide/keras/save_and_serialize#savedmodel_format.

"TensorFlow Lite Documentation." 2021, June. https://www.tensorflow.org/lite/performance/post_training_quantization.

"Tensorflow Repository." 2021, June. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/tensor_bundle/tensor_bundle.h.

"TensorFlow Source Code Documentation." 2021, June. https://www.tensorflow.org/jvm/api_docs/java/org/tensorflow/proto/util/BundleEntryProto.

"Turing-NLG: A 17-billion-parameter language model by Microsoft." 2020, February. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

Vartak, Manasi, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. "Mistique: A system to store and query model intermediates for model diagnosis," 1285–1300.

"Vertex AI." 2021, June. https://cloud.google.com/vertex-ai.

Wang, JunPing, WenSheng Zhang, YouKang Shi, ShiHui Duan, and Jin Liu. 2018. "Industrial Big Data Analytics: Challenges, Methodologies, and Applications." *arXiv e-prints,* arXiv–1807.

Wang, Sibo, Xiaokui Xiao, and Chun-Hee Lee. 2015. "Crowd-based deduplication: An adaptive approach," 1263–1277.

Wang, Yu Emma, Gu-Yeon Wei, and David Brooks. 2019. "Benchmarking tpu, gpu, and cpu platforms for deep learning." *arXiv preprint arXiv:1907.10701.*

"Wikipedia: Database." 2021, May. https://en.wikipedia.org/wiki/Database.

"Wikipedia: Distributed hash table." 2021, May. https://en.wikipedia.org/wiki/Distributed_hash_table.

"Wikipedia: Machine learning." 2021, May. https://en.wikipedia.org/wiki/Machine_learning.

"Wikipedia: Relational algebra." 2021, May. https://en.wikipedia.org/wiki/Relational_algebra.

"Wikipedia: Relational Database." 2021, May. https://en.wikipedia.org/wiki/Relational_database.

"Wolfram MathWorld: Tensor Rank." 2014, August. https://mathworld.wolfram.com/TensorRank.html.

Wu, Jiaojiao, Yanping Li, Tianyin Wang, and Yong Ding. 2019. "CPDA: A Confidentiality-Preserving Deduplication Cloud Storage With Public Cloud Auditing." *IEEE Access* 7:160482–160497. https://doi.org/10.1109/ACCESS.2019.2950750.

Wu, Wei, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. 2020. "A Review for Weighted MinHash Algorithms." *IEEE Transactions on Knowledge and Data Engineering.*

Wu, Yuncheng, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2021. "Serverless Model Serving for Data Science." *arXiv preprint arXiv:2103.02958.*

Xia, Wen, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. "A comprehensive study of the past, present, and future of data deduplication." *Proceedings of the IEEE* 104 (9): 1681–1710.

Yu, Sixing, Arya Mazaheri, and Ali Jannesari. 2020. "Auto Graph Encoder-Decoder for Model Compression and Network Acceleration." *arXiv preprint arXiv:2011.12641.*

Yuan, Binhang, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. "Tensor Relational Algebra for Distributed Machine Learning System Design." *Proc. VLDB Endow.* 14 (8): 1338–1350.

Zhou, Yanqi, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. 2020. "Transferable graph optimizers for ml compilers." *arXiv preprint arXiv:2010.12438.*

Zou, Jia, R Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. "PlinyCompute: a platform for high-performance, distributed, data-intensive tool development," 1189–1204.

Zou, Jia, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. "Lachesis: Automatic Partitioning for UDF-Centric Analytics."

Zou, Jia, Arun Iyengar, and Chris Jermaine. n.d. "Pangea: Monolithic Distributed Storage for Data Analytics." *Proceedings of the VLDB Endowment* 12 (6).