Serving Deep Learning Models with Deduplication from Relational Databases

Lixi Zhou
Jiaqing Chen
Amitabh Das
Arizona State University
(lixi.zhou,jchen501,adas59)@asu.edu

Hong Min
Lei Yu
IBM T. J. Watson Research Center
hongmin@us.ibm.com
lei.yu1@ibm.com

Ming Zhao
Jia Zou
Arizona State University
(mingzhao,jia.zou)@asu.edu

ABSTRACT

Serving deep learning models from relational databases brings significant benefits. First, features extracted from databases do not need to be transferred to any decoupled deep learning systems for inferences, and thus the system management overhead can be significantly reduced. Second, in a relational database, data management along the storage hierarchy is fully integrated with query processing, and thus it can continue model serving even if the working set size exceeds the available memory. Applying model deduplication can greatly reduce the storage space, memory footprint, cache misses, and inference latency. However, existing data deduplication techniques are not applicable to the deep learning model serving applications in relational databases. They do not consider the impacts on model inference accuracy as well as the inconsistency between tensor blocks and database pages. This work proposed synergistic storage optimization techniques for duplication detection, page packing, and caching, to enhance database systems for model serving. Evaluation results show that our proposed techniques significantly improved the storage efficiency and the model inference latency, and outperformed existing deep learning frameworks in targeting scenarios.

PVLDB Reference Format:
Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, and Jia Zou. Serving Deep Learning Models with Deduplication from Relational Databases . PVLDB, 15(10): 2230 - 2243, 2022.
doi:10.14778/3547305.3547325

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/asu-cactus/netsdb/tree/master/model-inference.

1 INTRODUCTION

In the life cycle of deep learning, serving models for inferences is a vital stage and usually incurs significant operational costs. An Amazon user study found that model serving is responsible for 45-65% of the total cost of ownership of data science solutions [7]. One important reason is that most of today's platforms that serve deep neural network (DNN) models, such as Nexus [64], Clipper [24], Pretzel [45], TensorFlow Serving [62], and Rafiki [70], are standalone systems that are totally decoupled from the data management systems. From the perspective of end-to-end applications, this decoupling incurs significant costs as follows:
(1) Existing deep learning serving frameworks are compute-focused and require each tensor fit in memory, otherwise the system fails. For large models with weight tensors [1], this problem significantly impacts the availability of a model serving system.
(2) The physical decoupling of data serving and model serving introduces management complexity and extra latency to transfer input features from the databases where input features are extracted to the deep learning frameworks.

Therefore, it is imperative to investigate the serving of deep learning models natively from the relational database management system (RDBMS) [14, 27, 36, 39, 40, 43, 61, 71, 74]. RDBMS has a long history of optimizing the memory locality, whether the working set size exceeds memory capacity or not, through effective buffer pool management. It also eases the management of data through data independence, views, and fine-grained authorization. All of these capabilities, if leveraged for model serving, will significantly reduce the operational costs and simplify system management for a broad class of real-world workloads [63], such as credit-card fraud detection, targeting recommendation, and conversational-AI for customer supports. In such applications, the features are extracted from various historical transaction records or customer profiles, which are stored in RDBMS.

Model deduplication in RDBMS for Serving. Managing multiple similar models at the serving stage, such as for model roll-back, versioning, personalization, A/B tests, and ensemble inference, has become a common pattern of DNN model serving [23, 24, 60]. Such DNN models contain abundant *similar* tensor blocks. Careful selection of similar tensor blocks for deduplication may not significantly affect the accuracy and may significantly reduce the storage space, memory footprint, and cache misses, and thus may reduce the inference costs and latency. However, existing deduplication techniques for tensors [68], files [12, 26, 48, 58, 69, 78], relational data [10, 13, 16, 28, 34, 72, 73], and MapReduce platforms [22, 41, 42], are not applicable to *model serving from RDBMS* because: (1) They do not consider the impacts on model inference accuracy; (2) They do not consider how existing database storage functionalities, including indexing, page packing, and caching, should be enhanced to better support the inference and the deduplication of DNN models.

The challenges that we focus on in this work include:

1. How to leverage indexing to efficiently detect similar parameters that can be deduplicated without hurting the inference accuracy?

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547325

2. A database page can contain multiple tensor blocks. How to pack tensor blocks into pages to maximize page sharing across multiple models and minimize the total number of needed pages for representing all tensors?
3. How to augment the caching policy to increase the data locality for deduplicated model parameters, so that pages that are needed by multiple models have a higher priority to be kept in memory?

To address these challenges, in this work, we propose a novel RDBMS storage design optimized for tensors and DNN inference workloads. Deep learning computations are mapped to relational algebra expressions [74]. A tensor is partitioned and stored as a set of tensor blocks of equivalent shape, where each block contains the metadata that specifies its position in the tensor. A tensor is similar to a relation and a tensor block is similar to a tuple. A DNN model inference is represented as a relational algebra graph, as detailed in **Sec. 2**. This high-level abstraction is also consistent with many popular systems that integrate database and machine learning, such as SystemML [14], Spark MLlib [57], SciDB [66], SPORES [71], LaraDB [36], among others.

Similar to the classical physical representation of a relation, we store a tensor as a set of database pages, with each page containing multiple tensor blocks. The difference is that each tensor relation consists of a set of private pages, and an array of references to shared pages that belong to more than one tensor, as detailed in **Sec. 3**. On top of such physical representation, we propose novel and synergistic indexing, paging, and caching techniques as follows:

Tensor block index for fast duplication detection (Sec. 4). It is widely observed that a small portion of model parameters (e.g., weights, bias) are critical to prediction accuracy. Deduplicating these parameters will lead to a significant reduction in accuracy [44]. To address the problem, different from existing tensor deduplication works [68], we propose to first measure each tensor block's sensitivity to prediction accuracy based on weight magnitude or other post-hoc analysis [33], and thus avoid deduplicating accuracy-critical blocks. Because pair-wise similarity-based comparison across tensor blocks exhibits inhibitive overhead, we used the Locality Sensitive Hash (LSH) based on Euclidean (L2) distance [37, 77], to facilitate the nearest neighbor clustering.

Packing distinct tensor blocks to pages for minimizing storage size (Sec. 5). The problem is a variant of the Set Basis problem [29] with a new constraint on the size of each set that belongs to the Set Basis (i.e, page size limit). To address this problem, we propose a concept called `equivalent class` so that blocks that are owned by the same set of tensors will be assigned to the same class. Then, we propose a two-stage algorithm that first packs tensor blocks in each equivalent class to pages respectively, and then repacks the tensor blocks from non-full pages.

Deduplication-aware buffer pool management (Sec. 6). Existing deduplication-aware cache replacement strategies [48, 69] do not consider the locality patterns of different sets of pages, which are important for model inference where the input/output of each layer have different locality patterns. However, existing locality-aware buffer pool management [21, 82, 83] do not distinguish private pages and shared pages. To address this problem, we propose a cost model for locality-aware page eviction, which gives pages that are shared by more tensors higher priority to be kept in memory.

The key contributions of our work are as follows:
1. We are the first to systematically explore the storage optimization for DNN models in RDBMS, with an overall goal of supporting deep learning model serving (i.e., inferences) natively from RDBMS.
2. We propose three synergistic storage optimizations: (a) A novel index based on L2 LSH and magnitude ordering to accelerate the discovery of duplicate tensor blocks with limited impacts on the accuracy; (b) A two-stage strategy to group tensor blocks to pages to minimize the number of pages that are needed to store all tensors; (c) A novel caching algorithm that recognizes and rewards shared pages across locality sets. It is noteworthy that our optimization can work together with other compression techniques such as pruning [32, 33] and quantization [38] to achieve a better compression ratio, as detailed in Sec. 7.6.2.
3. We implement the system in an object-oriented relational database based on our previous work of PlinyCompute [80–83], called netsDB [1]. We evaluate the proposed techniques using the serving of (1) multiple customized Word2Vec embedding models; (2) multiple versions of text classification models; (3) multiple specialized models for extreme classification; (4) multiple models of heterogeneous architectures. The results show that our proposed deduplication techniques achieved 2.7× to 3.6× reduction in storage size, speeded up the inference by 1.1× to 4.7×, and improved the cache hit ratio by up to 1.6×. The results also show that netsDB outperformed TensorFlow for these workloads.

2 BACKGROUND

2.1 Fundamentals of Deep Learning Inferences

A deep learning model usually consists of multiple layers. During the inference process, one layer's output will be the next layer's input features. We give two examples of layers: fully-connected layer and embedding layer, which are widely used in DNNs running on features extracted from relational data.

1. Fully-connected layer. The left part of Fig. 1 illustrates the example of a fully connected neural network (FFNN) that consists of multiple fully-connected layers. Each layer has a weight matrix, such as W_0, where each weight ($e_{i,j}$) is associated with an edge that connects one neuron (N_i) and one input feature (x_j). At a fully-connected layer, the weight tensor (e.g., W_0) is multiplied with the input feature vector (or a tensor that represents a batch of inputs) (X^T). The output is added to the bias vector ($bias$), and then applied with an activation function (σ), such as `ReLU` and `Sigmod`. Then the final output is sent to the next layer as input.

2. Embedding layer. An embedding layer [31] can be used to convert a token into an embedding vector. It is widely used in natural language processing, recommendation, etc. An embedding layer is usually stored as an $n \times d$ matrix, where n represents the size of the dictionary of words, and d represents the dimension size of a word embedding vector. As illustrated in Fig. 2, there are usually two approaches to look up an embedding for one token. One approach is to represent the token as a one-hot vector and multiply

[1] https://github.com/asu-cactus/netsdb. Related documentation can be found in https://github.com/asu-cactus/netsdb/tree/master/model-inference/.

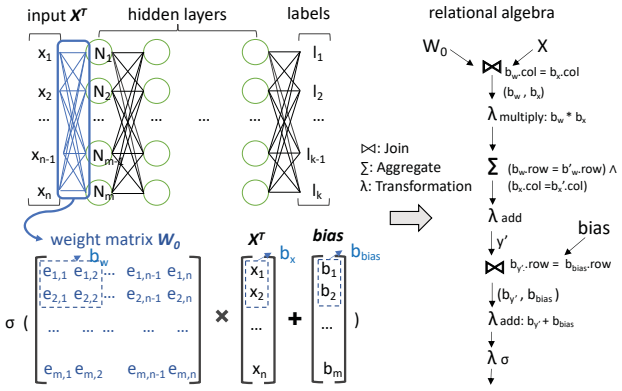

Figure 1: Illustration of Dense (Fully-Connected) Layers

the vector with the embedding matrix. The other approach is to use the index of the word in the dictionary to look up the embedding vector via filtering or indexing.

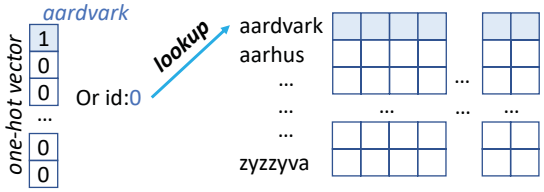

Figure 2: Illustration of the Word2Vec Embedding Layer

2.2 Inferences as Relational Queries

Existing works [14, 39, 51, 57, 74] propose to: (1) Abstract the tensor as a set of tensor blocks; (2) Encode local linear algebra computation logics that manipulate single or a pair of tensor blocks, in user defined functions (UDFs), also called as kernel functions, such as matrix multiplication, matrix addition, etc.; (3) Apply the relational algebra operators nested with these UDFs for performing linear algebra computations.

For example, **matrix multiplication** is a `join` followed by `aggregation` [14, 39, 51, 74]. The `join` pairs two blocks from the two tensors if the first block's column index equals the second's row index. Then each joined pair of tensor blocks is applied with a UDF that multiplies these two tensor blocks. An output block has its row index being the first block's row index and its column index being the second block's column index. Then all tensor blocks output from the transformation are `grouped by` their row and column indexes, and all tensor blocks in the same group will be added up in an aggregate/reduce UDF. Similarly, **matrix addition** is a `join`. In addition, as described in more detail in Tensor Relational Algebra (TRA) [74], other types of neural networks can also be represented in relational algebra. For example, **matrix transpose** is a `transform`; **activations** such as ReLU, tanh, and Sigmoid are `transforms`; **softmax and normalization** can be represented as an `aggregation` followed by a `transform`.

Therefore, as illustrated in Fig. 1, a fully-connected feed-forward network (FFNN) can be represented in relational algebra [39, 51].

Similarly, the two approaches of embedding lookup relying on vector-matrix multiplication and filtering can also be easily represented in relational algebra respectively.

3 SYSTEM OVERVIEW

Leveraging tensor relational algebra [39, 74], a tensor is represented as a set of tensor blocks. Without deduplication, the set is physically stored in an array of pages of equivalent size, where each page consists of multiple tensor blocks. With deduplication, certain pages will be shared by multiple tensors. These shared pages are stored separately in a special type of set. Each tensor not only stores an array of private pages, but also maintains a list of page IDs that points to the shared pages that belong to the set.

Given a set of models, we propose a novel **deduplication process**, as illustrated in Fig. 3 and described below:

(1) An LSH-based index is incrementally constructed to group tensor blocks based on similarity, so that similar tensor blocks can be replaced by one representative tensor block in their group, with limited impacts on the model inference accuracy. To achieve the goal, the main ideas include: (a) Always deduplicating the tensor blocks in the ascending ordering of their estimated impacts on the accuracy; (b) Periodically testing the deduplicated model inference accuracy along the duplication detection process, and stopping the deduplication for tensor blocks from a model, if its accuracy drops below a threshold. (Sec. 4) Validation datasets are often available at deployment stage, for pruning, fine-tuning, and handling concept drifts [47]. Such datasets can be reused for the periodical accuracy validation. However, we also provide an alternative approach that does not require validation datasets and relies on LSH parameter tuning to strike various trade-offs between accuracy and storage efficiency as discussed in Sec. 4.3.

(2) Each set of tensor blocks is physically stored as an array of pages of fixed size on disk. Distinct tensor blocks identified by the indexing are carefully grouped to pages so that each tensor is exactly covered by a subset of pages, and the number of pages that are required by all models is minimized. We optimize these objectives by assigning distinct tensor blocks that are shared by the same set of tensors to one equivalent class. Then blocks in the same equivalent class are grouped to the same set of pages. After this initial packing, tensor blocks from non-full pages are repacked to further improve the storage efficiency. (Sec. 5)

(3) The pages are automatically cached in the buffer pool. When memory resources become insufficient, the buffer pool manager will consider the locality patterns of each tensor and give hot pages and shared pages higher priority to be kept in memory. (Sec. 6)

4 INDEX FOR DUPLICATION DETECTION

4.1 Problem Description

In this section, we focus on the following problem: For a set of tensors that store model parameters, *which may have different shapes but are partitioned into tensor blocks that have the same shape,* how to divide all tensor blocks into distinct groups, so that blocks in each group can replace each other without a significant drop in the inference accuracy of each model? The problem is formalized as follows: Given k tensors:$T = \{t_1, ..., t_k\}$, the i-th tensor t_i is split into n_i tensor blocks: $t_i = \{b_1, ..., b_{n_i}\}$. The question is how to divide all tensor blocks, $B = \cup_i t_i$, into m clusters: $C = \{c_1, ..., c_m\}$, so that (1) $\forall c \in C, c \subset B$; (2) $\forall c_i, c_j \in C, c_i \cap c_j = \phi$; (3) $\forall c \in C$, $\forall b_i, b_j \in c, b_i \approx b_j$. Here, $b_i \approx b_j$ means that b_i can be replaced by b_j so that the drop in model accuracy is smaller than a threshold t.

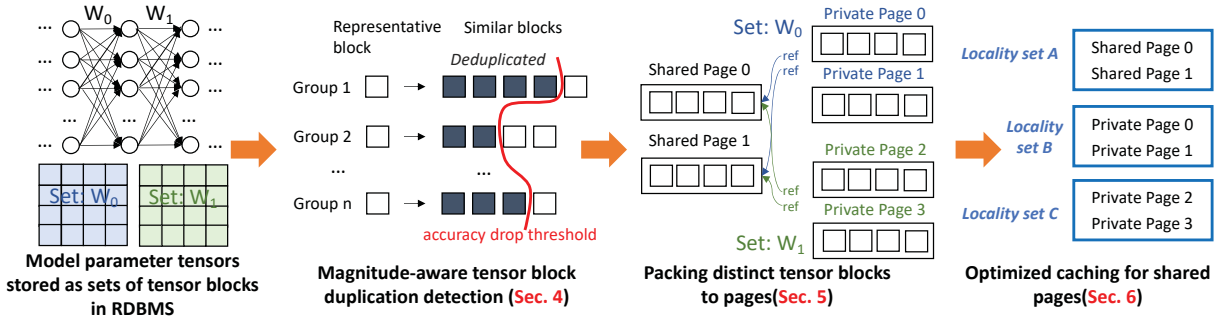

Figure 3: Overview of the proposed model deduplication workflow.

4.2 Main Ideas

4.2.1 Magnitude-aware Duplicate Detection. Existing works about deduplication [10, 13, 16, 22, 28, 34, 41, 42, 48] and tensor chunk deduplication such as Mistique [68] for model diagnosis, investigated exact page deduplication and similarity-based approximate page deduplication. However, we found these works cannot be directly applied to tensor block deduplication for model serving applications: (1) Exact deduplication of tensor chunks does not consider the fuzziness or similarity of model weights. . The number of tensor blocks that can be deduplicated based on exact match is 3× lower than similarity-based match as illustrated in Tab. 5. (2) We also found it *ineffective* to perform deduplication solely based on the similarity, without considering the impact of model weights on the prediction accuracy. For example, we found that deduplicating similar blocks in a batch normalization layer in a ResNet50 model (two blocks with less than 0.1% different weights were considered as similar), without considering the importance of weights, will reduce accuracy from 81% to 8%. Therefore, it is critical to develop new methods to identify tensor blocks that can be deduplicated with limited impacts on accuracy.

Motivated by the iterative pruning process [32, 33], in which weights with small magnitude are pruned first, we developed a process of magnitude-aware duplicate detection, where blocks of smaller magnitude are deduplicated first, and the model accuracy is periodically validated after deduplicating every k blocks.

4.2.2 LSH-based Approximate Tensor Block Deduplication. To reduce the pair-wise similarity comparison overhead, we consider leveraging Locality Sensitive Hash (LSH), which is a popular technique to solve nearest neighbor problems. LSH based on Hamming distance [25], Euclidean distance [37], and cosine similarity [19] are designed to identify similar numerical vectors with fixed dimensions, and can be directly applied to detect similar tensor blocks. In addition, the MinHash based on Jaccard similarity [17] is designed to identify similar binary vectors or similar sets of items. In this work, we mainly use the LSH based on Euclidean distance [20, 37], which we call L2 LSH, because it is easy to compute (e.g., it does not require an expensive numeric value discretization process like MinHash) and it can be linked to the JS-divergence [49] of weights' probability distributions of two tensor blocks [20].

4.3 Index Building

Given a set of models, for each model, we execute the steps as follows *for each model layer* ordered by its tensor size descendingly:

Step 1. Calculate an aggregated magnitude value (e.g., average, median, 1st quartile, 3rd quartile, etc.) for each tensor block in the tensors of the model layer. We use the 3rd quartile, because even if the block contains only a few large magnitude weights, it may impact the inference accuracy significantly and should not be deduplicated. The 3rd quartile can better reflect both the magnitude and quantity of large weights in a block than aforementioned alternatives, as illustrated in Fig. 4. The magnitude measurement can also be replaced by more complicated ones such as L2-norm, information measurements [52], etc.

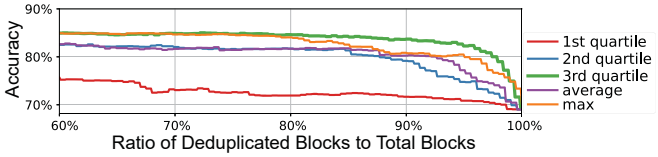

Figure 4: Comparison of different magnitude measurements when deduplicating an embedding layer pretrained using Wikipedia and a variant of the embedding finetuned using the IMDB datasets.

Step 2. Order all tensor blocks in the model by their magnitude values in ascending order.

Step 3. Select k blocks that have the lowest magnitude values, and for each block, its LSH signature is computed to query the index. If the index has seen similar blocks before, the block's identifier is added to the corresponding group and this block is replaced by the representative block, which is the first indexed block in this group. If the index hasn't seen a similar block, a new group is created, and this block becomes the representative block in the group.

Step 4. Test the model using a validation dataset to check whether its inference accuracy drop is less significant than a threshold t. If so, the algorithm repeats Step 3 and 4. Otherwise, it will *stop deduplication* for this model. That said, it simply adds each remaining block to the corresponding group, but such block will NOT be replaced by the representative block in the group. Such remaining blocks as well as the representative blocks are called as distinct blocks, each of which has one physical copy.

We repeat the above process for each layer of each model to incrementally construct the index, as illustrated in Alg. 1.

The output of the algorithm is $F_T = \{f_1, ..., f_k\}$. Each f_i is a mapping for the i-th tensor in the model, which specifies the identifier of the distinct tensor block corresponding to each (logical) block in

the tensor. The deduplication is achieved by allowing multiple tensor blocks across models mapped to one distinct block. The output is used in the page packing process as detailed in Sec. 5.

Algorithm 1 Index Building

INPUTS: $T = \{t_1, ..., t_k\}$ (A set of parameter tensors in a model layer), idx (The index that has been constructed for previous models, and will be updated by this layer), $L = \{d_1, ..., d_m\}$ (A set of distinct blocks derived from previous models, which will be updated by this layer)
OUTPUT: $F_T = \{f_1, ..., f_k\}$ (f_i maps a block in t_i to a distinct block)
$B = \{b_1, ..., b_n\} \leftarrow \cup_{i=1}^{k} t_i$
$a_0 \leftarrow accuracy(Model_B)$; $i \leftarrow 0$
$B' = \{b'_1, ..., b'_n\} \leftarrow$ sort B by the magnitude of $b_i \in B$ ascendingly
while $i \leq n$ **do**
  for $j = i + 1, ..., i + k$ **do**
    $s_j \leftarrow lsh(b'_j)$
    if idx.count$(s_j) > 0$ **then**
      $(b_c, c) \leftarrow idx$.look_up(s_j);
      $c \leftarrow \{(tensorID(b'_j), blockID(b'_j))\} \cup c$
      idx.update$(s_j, (b_c, c))$
      $b'_j \leftarrow b_c$ //use representative block b_c to replace b'_j
      $f_{tensorID(b'_j)}[blockID(b'_j)] \leftarrow IndexInL(b_c)$
    else
      idx.insert$(< s_j, (b'_j, \{(tensorID(b'_j), blockID(b'_j))\} >)$
      L.push_back(b'_j)
      $f_{tensorID(b'_j)}[blockID(b'_j)] \leftarrow IndexInL(b'_j)$
    end if
  end for
  $a \leftarrow accuracy(Model_B)$
  if $a_0 - a > t$ **then**
    for $u = j + 1, ..., n$ **do**
      idx.insert$(< lsh(b'_u), (b'_u, \{(tensorID(b'_u), blockID(b'_u))\} >)$
      L.push_back(b'_u)
      $f_{tensorID(b'_u)}[blockID(b'_u)] \leftarrow IndexInL(b'_u)$
    end for
    return F_T
  end if
  $i \leftarrow i + k$
end while
return F_T

Fine-Tuning. In order to further improve the accuracy, after deduplicating the models based on the constructed index, an additional parameter finetune stage can be carried out to optimize the accuracy after deduplication. In our implementation, for simplicity, during the finetune process, the tensor blocks that are shared by multiple models will be frozen, and only the weights in the private pages will be tuned for each model.

Weight Normalization. Normalization is not helpful for layer-wise deduplication (i.e., each iteration of Alg. 1 takes tensor blocks in a layer as input). That's because tensor blocks in one layer will be ordered and deduplicated together, separated from other layers. Our experiments also showed that both cross-layer and intra-layer normalization can hardly affect the effectiveness of the layer-wise deduplication.

Alternative Approach to Periodical Accuracy Validation. The periodical accuracy validation in Alg. 1 will bring storage and latency overheads, as discussed and evaluated in Sec. 7.3.1. Such overheads can be avoided by an alternative approach that fully relies on the tuning of the LSH collision threshold [2]. It means that all tensor blocks that have matches in the index will be deduplicated without validation of accuracy. But the users can tune the collision threshold to control the trade-off between accuracy and storage efficiency. We evaluate this alternative approach in Sec. 7.3.1.

5 GROUPING TENSOR BLOCKS INTO PAGES

Based on Sec. 4, we obtained a mapping from each (logical) tensor block to a (physical) distinct block. Each tensor may consist of both private distinct blocks that belong to only one tensor and shared distinct blocks that belong to multiple tensors. Now we investigate the problem of how to pack multiple tensor blocks to database pages, so that we can maximize the sharing of pages and minimize the total number of pages that are needed.

Database storage organizes data in pages, so that a page is the smallest unit of data for I/O read/write and cache load/evict operations. Analytics databases usually use a page size significantly larger than a tensor block (e.g., Spark uses 128 megabytes page size and 1024×1024 block shape by default [57]). As a result, a database page may contain multiple tensor blocks. Each tensor consists of a set of pages that should contain exactly the set of tensor blocks belonging to the tensor: no more and no less. If these pages contain tensor blocks that do not belong to the tensor, it will significantly complicate the scanning and various operations over the tensor.

However, the default paging process used in database systems cannot work well with deduplication. By default, tensor blocks are packed into pages based on the ordering of the time when each block is written to the storage. If a page can hold up to l tensor blocks, every batch of l consecutive tensor blocks is packed into one page. However, in such default packing, private (e.g., block 17-20 in Fig. 5) and shared tensor blocks (e.g., block 1-16) may get packed to the same page. Such a page cannot be deduplicated because of the private tensor blocks. As illustrated in Fig. 5, after performing deduplication, so that each distinct page will be physically stored once, the default packing requires 8 pages, while a better packing scheme requires only 5 pages.

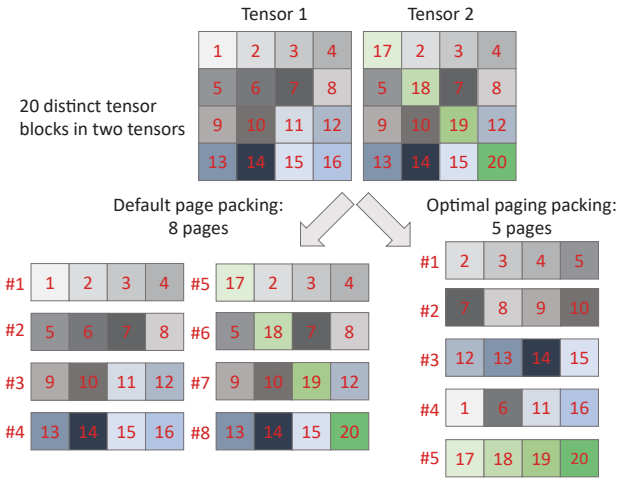

Figure 5: Motivation of page packing optimization

[2] An LSH signature is usually split to multiple bands, and the collision threshold is the minimal number of matching bands required for a match of two LSH signatures.

5.1 Problem Formalization

The problem is: *How to group the tensor blocks across all models to pages to satisfy that: (1) For each tensor, we can find a subset of pages so that the set of tensor blocks contained in the pages is exactly the set of all tensor blocks that belong to the tensor; (2) The total number of distinct pages that need to be physically stored is minimized.*

Here we formalize the decision problem corresponding to the above optimization problem, called as multi-tensor page packing decision problem (MTPPDP), as following:

Input: A finite set of distinct tensor blocks $B = \{b_1, ..., b_n\}$, and a set of tensors $T = \{t_1, ..., t_k\}$, and a page size limit l. (Each tensor t_i is a set of tensor blocks, i.e., $t_i \subset B$.)
Question: Does there exist a collection of pages $P = \{p_1, ..., p_s\}$, such that (1) $p_i \subset B$; (2) each page has no more than l blocks, denoted as $|p_i| \leq l$; and (3) for each tensor $t_i \in T$, there exists a subcollection of s pages, whose union is exactly t_i, denoted as $P' \subset P$, so that $\cup_{p_j \subset P'} p_j = t_i$.

It is an important problem, because large page sizes up to hundreds of megabytes, are widely adopted in analytics databases [75] and when memory resources become insufficient, even saving only a few pages may significantly improve the performance.

Theorem 1. ***MTPPDP is NP complete.***

Proof. The *Set Basis decision problem* [29, 65], which is NP complete, can be reduced to MTPPDP in polynomial time. The *Set Basis decision problem* is defined as follows:

Given a collection D of subsets of a finite set S, positive integer $n \leq |D|$, the decision problem is to determine whether there exists a collection I of n subsets of S ($|I| = n$), such that, for each $d \in D$, there is a subcollection of I whose union is exactly d.

The *Set Basis decision problem* can be reduced to MTPPDP in polynomial time as follows: (1) $B = S$, (2) $T = D$.If the MTPPDP problem has a solution P, P is also a solution to the Set Basis decision problem. If the Set Basis decision problem has a solution I, we can obtain a solution of P for the MTPPDP problem by breaking every subset whose size is larger than the page size limit l into multiple smaller subsets, so that each subset's size limit is smaller than l. Therefore, Theorem 1 is proved.

In particular, the related optimization problem, which is to search for a *minimal* collection of pages P that satisfies the conditions, is NP-hard: (1) The problem is at least as hard as the corresponding decision problem, which is NP complete [29]; (2) There is no known polynomial-time verification for a solution of the problem.

There exist greedy algorithms to solve the Set Basis optimization problem, which choose from the basis candidate sets, constructed from the intersections of sets in S [30, 67]. These algorithms cannot be applied to our MTPPDP problem, because these algorithms do not apply size constraints for each set in the set basis B.

5.2 A Two-Stage Page Packing Strategy

To solve the optimization problem, we first propose to group tensor blocks into **equivalent classes**. Different tensor blocks that are shared by the same set of tensors are assigned to the same equivalent class, as illustrated in Fig. 6, which depicts the tensor sharing relationship for the example in Fig. 5. It is beneficial to use a divide and conquer strategy to pack for each equivalent class in parallel by grouping the blocks falling into the same equivalent class to the same page(s). That's because these pages can be shared by all tensors associated with the page's corresponding equivalent class. By doing so, in the above example (Fig. 5), the 12 distinct blocks in equivalent class C_3 are packed to three pages, the four distinct blocks in C_1 are packed to one page, and the four distinct blocks in C_2 are packed to one page, which leads to the optimal plan for this case, as shown in Fig. 6. The algorithm is illustrated in Alg. 2.

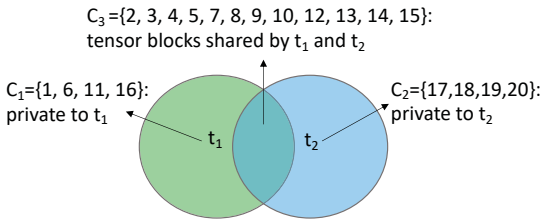

Figure 6: Illustration of equivalent classes of tensor blocks for page packing for the example in Fig. 5.

Algorithm 2 Equivalent Class-Based Greedy Strategy

```
INPUTS: T, B, l
OUTPUT: P
{C_1, ..., C_m} ← B, T {divide B into multiple equivalent classes, so blocks in each class are shared by the same set of tensors}
P ← φ
for i=0..m do
    p ← φ
    for b : C_i do
        if |p| < l then
            p ← p ∪ {b}
        else
            P ← P ∪ {p}; p ← φ
        end if
    end for
    if |p| > 0 then
        P ← P ∪ {p}; p ← φ
    end if
end for
return P
```

The problem with the equivalent class-based packing is that it may lead to non-full pages, because items in certain equivalent classes may not fully fill the pages. For example, as illustrated in Fig. 7, if a page can maximally hold two blocks, the blocks in C_1, C_2, C_6 will be packed to three non-full pages respectively. However, a better scheme is to pack these blocks into two pages: $p_1 = C_1 \cup C_6$ and $p_2 = C_1 \cup C_2$. Therefore, we propose a two-stage strategy for optimizing page packing schemes. At the first stage, blocks from each equivalent class are packed to pages separately, and no page is allowed to mix blocks from different non-equivalent classes. Then, we run the second stage by repacking blocks from non-full pages, and applying an approximation algorithm based on the following heuristics: (1) Largest-Tensor-first. A tensor that contains more tensor blocks to be repacked is more likely to generate pages that can be reused by other tensors. (2) Hottest-Block-First. Frequently

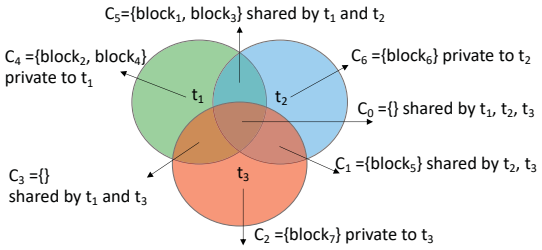

Figure 7: Another example: the equivalent class-based greedy strategy leads to three non-full pages for C_1, C_2, C_6.

shared tensor blocks, if packed together, are more likely to generate pages that can be reused across multiple tensors.

The approximation algorithm picks the tensor that has the most tensor blocks in non-full pages to repack first. When it repacks for a given tensor, it first attempts to identify and reuse packed pages that cover as many blocks to repack as possible. Then it orders the remaining tensor blocks first based on their sharing frequency (i.e., the number of tensors a block is shared by), and the ordering of their equivalent classes. Then, it packs these blocks into pages in order, without leaving any holes in a page except for the last page. We formalized the algorithm for the second stage as Alg. 3. (The algorithm for the first stage is the same with Alg. 2.) Alg. 3 is scalable to a large number of equivalent classes, so it can also be used independently for a large number of tensors. In Sec. 7.4, we compare the performance of only using Alg. 3, only using Alg. 2, and the two-stage algorithm.

5.3 Algorithm Analysis

Here, we use $Alg2(P)$ and $TwoStage(P)$ to denote the solution size of Alg. 2 and the Two Stage algorithm, and $OPT(P)$ to denote the optimal solution size. l refers to the maximal number of blocks in one page. k refers to the number of tensors.

Theorem 2. $Alg2(P) \leq OPT(P) + 2^k - 1$

Proof. First, $OPT(P) \geq \lceil |\cup t_i|/l \rceil$. That's because every page has at most l blocks, and we have in total $|\cup t_i|$ blocks, so we have at least $\lceil |\cup t_i|/l \rceil$ pages.

Second, in Alg. 2, tensor blocks from each equivalent class C_i are packed to pages separately. Each C_i can be divided into two disjoint sets: $C_i^{(1)}$ and $C_i^{(2)}$, so that: (1) $|C_i^{(2)}| = |C_i|\%l$ and the blocks in $C_i^{(2)}$ will be packed to at most one non-full page; and (2) $C_i^{(1)} = C_i - C_i^{(2)}$ and the blocks in $C_i^{(1)}$ will be packed to full pages because $|C_i^{(1)}|\%l = 0$. Because there are at most $2^k - 1$ equivalent classes, and each equivalent class has at most one non-full page, $Alg2(P) \leq |\cup C_i^{(1)}|/l + 2^k - 1$. Because $|\cup C_i^{(1)}|/l \leq \lceil |\cup t_i|/l \rceil$, we have $|\cup C_i^{(1)}|/l \leq OPT(P)$. Therefore, we proved $Alg2(P) \leq OPT(P) + 2^k - 1$.

In practice, we found the second stage (Alg. 3) is mostly helpful when there are a large number of equivalent classes and each has only a few remaining blocks. If every equivalent class has at most u remaining blocks ($u < l$), we will have $TwoStage(P) \leq OPT(P) + \lfloor \sum_{i=1}^{k} N_{(k)}^{(i)} \times (i-1) \times u/l \rfloor + k$. Here, $N_{(k)}^{(i)}$ denotes the number of equivalent classes that are associated with i of total k tensors. The blocks in such equivalent classes will have at most $i-1$ additional copies using Alg. 3, depending on how frequently each page can be reused. Also, each tensor will lead to at most one non-full page, so we added k in the above formulation.

Algorithm 3 Approximation Strategy

```
INPUTS: T (A set of tensors for packing to pages. When applied to
  Stage-2, each tensor only contains blocks from non-full pages resulting
  from Stage 1), l, P (The set of pages that have been packed in Stage 1,
  when applied to Stage-2. P = φ if there is no Stage 1. More pages may
  be appended to the set during processing.)
OUTPUT: P (a set of pages as the final output)
T ← orderByNumTensorBlocksDescend(T)
for i = 1, ..., k do
    P' ← a set of existing pages belonging to P that form a maximal
    subset of t_i; I_δ ← t_i − ∪_{p∈P'} p
    if I_δ = φ then
        continue
    end if
    {b_1, ..., b_δi} ← orderBySharingFreqDescend(I_δ); p ← φ
    for j = 1, ..., δi do
        if |p| < l then
            p ← p ∪ {b_j}
        else
            P ← P ∪ {p}; p ← φ
        end if
    end for
    P ← P ∪ {p}; p ← φ
end for
return P
```

6 BUFFER POOL MANAGEMENT

An important factor in buffer pool management is to estimate the probability that a page will be reused again within the next t time ticks, denoted as p_{reuse}. Widely used page replacement algorithms such as LRU/MRU/LFU mainly consider p_{reuse} and use reference time, distance, and frequency to model it.

Our previous works in locality-set-based page eviction [82, 83], consider more factors in addition to p_{reuse} by modeling the eviction costs as Eq. 1, which is a standard representation that considers cost for writing out a page (c_w) and the cost for loading a page back for reading (c_r) separately [50, 82, 83]. The idea is that the pages that will be accessed similarly (e.g., pages in one equivalent class) are regarded as a separate locality set. Each locality set will be configured with its own page eviction policy, e.g., MRU or LRU. When pages need to be evicted from the buffer pool to make room for new pages, the system chooses a locality set to be the victim if the next page-to-be-evicted from the locality set has the lowest expected eviction cost among all locality sets. Then one or more pages will be evicted from the victim using its own eviction policy. This algorithm has been proved to have better performance than LRU/MRU/LFU for workloads that have predictable locality patterns such as model serving [82, 83], where the computations' data access patterns at each layer are mostly known.

$$c_w + p_{reuse} \times c_r \tag{1}$$

However, these existing algorithms did not consider page sharing caused by model deduplication in the multi-model serving scenario. To address the problem, we propose to refactor the formulation

of p_{reuse}. We apply the queueing theory [35] to model the page accesses so that each page is like a server, and each model inference request that triggers a page access is like a customer. Because a page may be shared by multiple models, inference requests from each model will be dispatched to a queue associated with the model. If we assume the arrival time of the next access to each page from each queue as an independent Poisson point process [35], p_{reuse} can be estimated using Eq. 2. Here, $M = \{m_1, ..., m_k\}$ represents a set of models that share this page, and λ_i denotes the access rate per time tick for the model m_i.

$$p_{reuse} = 1 - e^{-\sum_{m_i \in M} \lambda_i t} \quad (2)$$

This approach is more accurate than simply estimating p_{reuse} based on the reference time/frequency/distance measured for each page, because the access patterns of various datasets involved in each model inference is fixed, mostly affected by λ_i.

We implemented the enhanced locality-set-based page eviction (using Eq. 2) in netsDB. Our evaluation results in Sec. 7.5 showed up to 1.6× improvement in cache hit ratio compared to MRU, LRU, and the original locality-set-based page eviction.

7 EVALUATION

7.1 Evaluation and Workloads

7.1.1 Multiple Versions of Personalized Text Embedding Models. A text embedding used for natural language processing is usually trained using a large open corpus like Wikipedia [6]. However, at the same time, every enterprise or domain has its own terminologies, which are not covered in the open data. To personalize the text embeddings, for each domain, we need to train the model on both the shared open data and the private domain/enterprise data. Therefore, we used a Word2Vec embedding downloaded from TFHub [5], which is pretrained using a Wikipedia dump. The model embeds about 1 million words. Each word corresponds to a 500 dimensional embedding vector. Therefore, the Word2Vec embedding layer has about one million 500 dimensional embedding vectors stored in a $1,009,375 \times 500$ weight tensor. Then we finetune the pre-trained model using different domain-specific corpus including texts extracted from Shakespeare's plays [4], posts collected from Firefox support forum [8], articles collected from Fine Wine Diary [8], Yelp reviews [76], IMDB reviews [53].

7.1.2 Multiple Versions of Text Classification Models. We further investigate a scenario that serves five different text semantic classification models. Each classification task takes a review as input and outputs a binary label to indicate the input is toxic or nontoxic [15, 53, 76]. All tasks use the same model architecture. Each model uses three layers. The first layer is a Word2Vec layer as mentioned in Sec. 7.1.1, using a vocabulary size of 1, 009, 375 and an embedding dimension of 500. The second layer is a fully connected layer that consists of merely 500×16 parameters, and the third layer is an output layer that consists of 16×2 parameters. Because the fully connected layers are small in size, we encode these in a UDF that is applied to the output of the Word2Vec embedding layer.

The first two text classification models are trained using the same Yelp datasets. The difference is that Model-1's embedding layer uses the weights of a pre-trained model directly downloaded from TFHub as mentioned in Sec. 7.2.1, which is set as `Non-Trainable`, so that only the weights of the fully connected layers are changed during the training process. However, Model-2's Word2Vec layer is set to be `Trainable`, which means the weights of the layer will also change during the training process. Similarly, Model-3 and Model-4 are trained using IMDB review datasets, with the embedding layer set to be `Non-Trainable` and `Trainable` respectively. The Model-5 is trained using the civil comments [15], collected from news sites, and its embedding layer is set to be `Trainable`.

7.1.3 Transfer Learning of Extreme Classification Models. Following TRA [74], a two-layer feed-forward neural network (FFNN) is implemented in netsDB for the AmazonCat-14K [55, 56] workload. This FFNN requires five parameter tensors: the weight tensors and bias tensors of the two layers, and the input tensor for which predictions are generated. The input tensor includes $1,000$ data points that have $597,540$ features, and the extreme classification task uses $14,588$ labels. The hidden layer has $1,000$ neurons. Therefore, the weight tensor (denoted as W_1) in the first layer has $597,540 \times 1000$ parameters, and the weight tensor (denoted as W_2) in the second layer has $14,588 \times 1000$ parameters.

A transfer learning scenario is tested, where the first layer W_1 is freezed, and W_2 is specialized for different tasks. Only for this scenario, the inputs, weights, and biases are randomly generated instead of being trained from real-world data like other scenarios. The experiments are still reasonable as deduplication in this scenario hardly affects the inference accuracy. That is because W_1 used in all the models are the same and thus no weights need to be approximated for deduplicating it, and we also choose not to deduplicate any blocks from the specialized and smaller W_2 layer.

7.1.4 Heterogeneous Models. We further investigate the deduplication of multiple models that have heterogeneous architectures.

Heterogeneous Scenario-1. In this scenario, we used four text classification models with different shapes of pre-trained embedding layers downloaded from TF-Hub. The first model, called as nnlm128_yelp [2, 11], is trained on the Yelp dataset with an embedding layer that has a dictionary size of $963,812$ and each embedding vector has a dimension of 128. Thus the shape of the embedding layer is $963,812 \times 128$. The second model, called as nnlm50_imdb [3, 11], is trained on the IMDB dataset with an embedding layer of the shape of $963,812 \times 50$. The third model, called wiki250_civil_comment [9, 59], is trained on the civil comment dataset with an embedding layer of the shape of $1,009,375 \times 250$. The fourth model, called wiki500_yelp is trained on the Yelp dataset, which is also used in Section 7.1.2. Its embedding layer has a shape of $1,009,375 \times 500$.

Heterogeneous Scenario-2. In this scenario, we used four extreme classification models as FFNN with different sizes for the input layer, hidden layer, and output layer. The first model is trained on RCV1-2K [46], and its input layer has $47,236$ features, its hidden layer has $5,000$ neurons, and its output layer has $2,456$ labels. The second model, is trained on the AmazonCat-13K dataset [54], and its number of features, hidden neurons, and labels are $203,882$, $1,000$, and $13,330$ respectively. The third model is trained on AmazonCat-14K [55, 56], which is described in Section 7.1.3. The fourth model is trained on EURLex-4.3K [18] and it has $200,000$ features, $2,000$ hidden neurons, and $4,271$ labels.

Heterogeneous Scenario-3. In this scenario, we investigate the deduplication of one text classification model wiki500_yelp from Scenario-1 and one extreme classification model AmazonCat-13K from Scenario-2.

Evaluation Environment Setup Unless explicitly specified, most of the experiments used an AWS r4xlarge instance that has four vCPU cores and 30 gigabytes RAM. The storage volumes include a 128 GB SSD, and a 128 GB hard disk drive. For the experiments on the GPU, we used an AWS g4dn.2xlarge instance that is installed with one NVIDIA T4 Tensor Core GPU that has 16 gigabytes memory, besides eight CPU cores and 32 gigabytes host memory.

The default buffer pool size is half of the available memory to balance caching and execution. We configure it to different values to compare the performance of the proposed approach and baselines with different levels of memory resources allocated for caching the model parameter tensors, the input feature tensors, etc.

7.2 Overall Evaluation Results

7.2.1 Multiple Versions of Personalized Text Embeddings. We find that word embedding models finetuned from the same TFHub pretrained Word2Vec model share more than 90% of pages. (The accuracy of each embedding model after finetuning is above 99%.) Each model is a $1,009,375 \times 500$ tensor, stored in a set of tensor blocks in the shape of $10,000 \times 100$, each weight is stored in double precision. Each input matrix is of the shape of $100 \times 1,009,375$, representing a batch of 100 words. It will multiply with the embedding matrix of the shape $1,009,375 \times 500$Without our proposed deduplication mechanism, storing six word embedding models separately requires more than 24 gigabytes storage space. However, by applying our work, only 6.7 gigabytes of storage space is required, which is a 3.6× reduction. Note that the overall memory requirements for serving 6 models will be higher than the storage requirements, as we also need to cache the intermediate data, which includes the `join` HashMap constructed for probing the model parameters, and about 1 gigabytes input data.

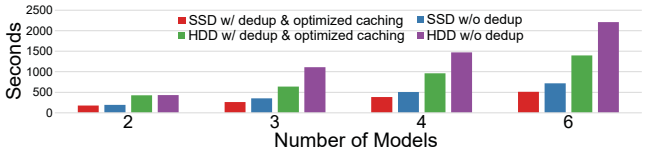

Figure 8: Overall latency for serving different number of Word2Vec models, tested in a r4xlarge instance, using SSD and HDD. Buffer pool size is set to 15 gigabytes.

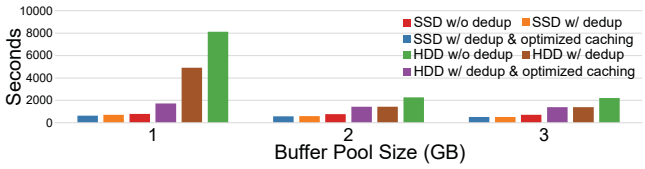

Figure 9: Overall latency for serving six word2vec models using different storage configurations

In Fig. 8 and Fig 9, we measured the total latency of making a batch of 100 inferences on all six models using different configurations for buffer pool size and storage hardwares. We observed that our proposed deduplication mechanism brought up to 1.4× and 4.7× speedups in model serving latency for SSD and HDD storage respectively, as illustrated in Fig. 8 and Fig. 9.

We also compared the netsDB's performance to the CPU-based TensorFlow on the same AWS r4.xlarge instance and the GPU-based TensorFlow on a g4dn.2xlarge instance. On TensorFlow, we developed two approaches for Word2Vec inference.

The first approach used matrix multiplication (`tf.matmul`), similar to netsDB's implementation. In the experiments of comparing this approach and netsDB, we used double precision for both systems. We still use the input batch size of 100 to be consistent with all above experiments.

The second approach is based on embedding lookup by using Keras' Word2Vec embedding layer (i.e., `keras.layers.Embedding`). The implementation takes a list of IDs as input, and searches the embedding for each ID (via index) in parallel.

For the second approach, because Keras' embedding layer enforces single precision, we changed netsDB implementation to use the single-precision float type. The experiments for this approach used 1 million IDs in each batch. We assume the 1 million IDs are from 100 documents, and each document has 10,000 different words, so its input features include 100 vectors, each vector is a sum of the one-hot embedding vectors of 10,000 words. Therefore, the input batch has 800 megabytes in size for the implementation based on matrix multiplication, but only 8 megabytes for the implementation based on embedding lookup.

In Tab. 1, TF-mem, TF-file, and TF-DB load an input batch from the local memory, the local CSV file, and a PostgreSQL table (400 BLOB fields for the first approach, and 1 BLOB field for the second approach), respectively. We observed that netsDB supports the inference of significantly more models in the same system than TensorFlow. For this case, we did not observe performance gain brought by GPU acceleration in TensorFlow, mainly because inference is less complicated than training and it cannot fully utilize the GPU parallelism and the benefits cannot outweight the overheads of moving data between CPU and GPU.

When all models fit to memory, TensorFlow has better performance than netsDB. That's because RDBMS introduces additional overheads such as constructing a hash map for the hash join as part of matrix multiplication, join-fork parallelism, query optimization and compilation, etc. However, such overheads can be avoided through materialization of hash map, asynchronous scheduling, and ahead-of-time query compilation, while preserving the benefits of the scalability brought by blocked tensors and relational processing. We will investigate this in our future works.

7.2.2 Multiple Versions of Text Classification Models. Based on the above results, we further evaluated the proposed techniques on the text classification task described in Sec. 7.1.2.

We imported these text classification models into netsDB. The default page size used in this experiment is 64 megabytes and when using a block shape of 100×10000, each text classification model requires 64 pages of storage size before deduplication. We first compared the required number of private and shared pages after

Table 1: Comparing the serving performance of multiple word2vec models deployed in netsDB to TensorFlow. (Unit: Seconds)

		TensorFlow CPU			TensorFlow GPU		
numModels	netsDB	TF-mem	TF-file	TF-DB	TF-mem	TF-file	TF-DB
Matrix-Multiplication-based inference, double precision							
3	252	9	64	96	14	69	128
6	503	Failed	Failed	Failed	Failed	Failed	Failed
12	1008	Failed	Failed	Failed	Failed	Failed	Failed
Embedding-lookup-based inference (1 million IDs/batch), single precision							
3	114	57	58	58	Failed	Failed	Failed
6	229	Failed	Failed	Failed	Failed	Failed	Failed
12	456	Failed	Failed	Failed	Failed	Failed	Failed

deduplication as well as the classifier inference accuracy before and after deduplication. The comparison results are illustrated in Tab. 2.

Without deduplication, the total storage space required is 20.5GB for 320 pages. After applying the proposed deduplication, the total storage space required is reduced to 5.6GB for 87 pages.

Table 2: Pages deduplicated (shared pages) and inference accuracy before and after deduplication.

	private pages	num shared pages	auc before dedup	auc after dedup
Model-1	2	62	85.01%	85.01%
Model-2	13	51	90.38%	86.79%
Model-3	7	57	81.25%	81.25%
Model-4	1	63	84.69%	81.11%
Model-5	1	63	94.80%	94.09%

The comparison of the overall inference latency of all five text classification models, using different block sizes and storage configurations, is illustrated in Fig. 10. We observed that 1.1× to 1.6× speedup were achieved by applying our proposed techniques.

7.2.3 Transfer Learning of Extreme Classification Models. In this experiment, all three models have the same architecture as described in Sec. 7.1.3, using double precision weights, and are specialized from the same feed-forward model through transfer learning and they share a fully connected layer, which contains 597 millions of parameters. This layer is stored as a shared set in netsDB, and it accounts for 4.8 gigabytes of storage space. Each model's specialized layer only accounts for 0.2 gigabytes of storage space. Therefore, with deduplication of the shared layer, the overall required storage space is reduced from 15 gigabytes to 5.4 gigabytes. We need to note that the required memory size for storing the working sets involved in this model-serving workload is almost twice of the required storage space, considering the input batch of the $1,000$ $597,540,000$ dimensional feature vectors and the intermediate data between layers for both models.

Besides a significant reduction in storage space, we also observed up to 1.18× and 1.45× speedup in SSD and HDD storage respectively, because of the improvement in cache hit ratio (40% – 46%), as illustrated in Fig. 11. Because this is a transfer learning scenario, the shared pages have no approximation at all, there exists no influence on accuracy.

We also compared the netsDB performance to TensorFlow, using the Keras implementation of the FFNN model. As illustrated in Tab. 3, netsDB outperforms TensorFlow for loading input from a CSV file and a Blob field of a PostgreSQL table. If we compute and store the input feature vectors in a table of 400 Blob fields, the TF-DB latency for CPU and GPU is $1,274$ and 945 seconds respectively, significantly slower than the latency on netsDB, which serves data and model in the same system.

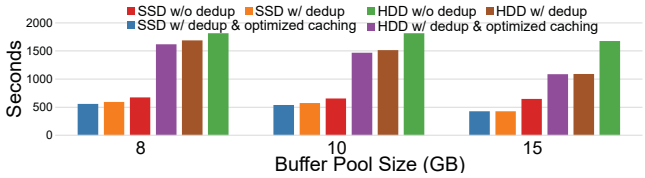

Figure 10: Overall latency for serving text classification models using different storage configurations.

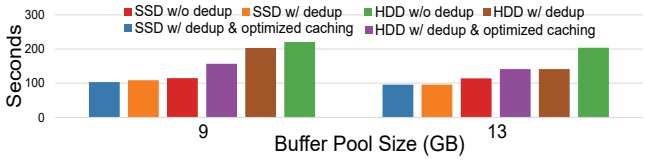

Figure 11: Overall latency for transfer learning with FFNN.

Table 3: Comparing the serving performance of multiple FFNN models deployed in netsDB to TensorFlow. (Unit: Seconds)

		TensorFlow CPU			TensorFlow GPU		
numModels	netsDB	TF-mem	TF-file	TF-DB	TF-mem	TF-file	TF-DB
2	64	43	383	94	17	310	55
3	96	64	*Failed*	115	*Failed*	*Failed*	*Failed*

7.2.4 Models of Heterogeneous Architectures. As illustrated in Tab. 4, our proposed approach achieve significant benefits in compression ratio and execution time speedup even for deduplicating models that have heterogeneous architectures, as described in Sec. 7.1.4. We also compare the maximum accuracy drop of all heterogeneous models involved in each scenario after applying our proposed deduplication approach. We used a page size of 64 megabytes, and the overall storage size has been reduced by 2.6× for scenario-1, 1.2× for scenario-2, and 1.8× for scenario-3. Despite the overheads in mapping each distinct block to its actual position in each tensor based on the block metadata when handling heterogeneous model architectures, we still observed 1.1× to 1.7× execution time speedup after applying the deduplication, due to the aforementioned reduction in memory footprint. Taking scenario-1 as example, nnlm128_yelp, wiki250_civil_comment, and wiki500_yelp achieved 1.3×, 2.3×, and 2.0× speedup in execution time respectively, and nnlm50_imdb runs 9% slower after deduplication. This showed that the speedup is positively correlated to the model size. In Scenario-1 we found 17% blocks are shared 2 to 100 times within one tensor, while this ratio is only 1% and 6% in Scenario-2 and 3. Such block will be stored once but mapped to multiple blocks in one tensor at runtime.

Table 4: Deduplication of Heterogeneous Model Architectures with 15GB buffer pool and SSD (block size: 50×10000)

Models	Blocks w/o dedup	Blocks w/ dedup	Maximum Accuracy Drop	Pages Needed w/o dedup	Pages Needed w/ dedup	Execution Time Speedup
Scenario-1	1922	514	3.77%	138	53	1.7×
Scenario-2	3625	2868	3.75%	238	194	1.1×
Scenario-3	1704	895	3.59%	114	63	1.2×

7.3 Evaluation of Duplicate Block Detection

We compared our indexing strategy as illustrated in Alg. 1 to two baselines: (1) A naive indexing scheme using pair-wise comparison

to identify similar blocks based on Euclidean distance; (2) Mistique's approximate deduplication using MinHash [68]. As illustrated in Fig. 12, we observed significant accuracy improvement brought by our proposed deduplication detection approaches (w/ and w/o fine-tune) for deduplicating the same amount of blocks. That's because both baselines failed to consider a block's magnitude as well as its impact on accuracy.

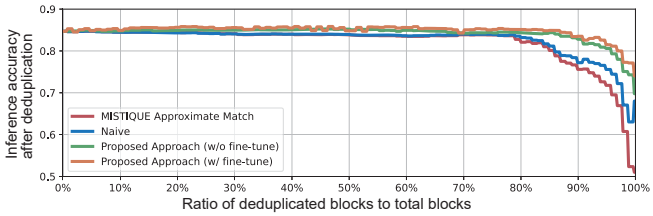

Figure 12: Comparison results of deduplicating a text classification model using different indexing approaches (block size: 100**x**10000**)**

Moreover, we also compared the compression ratio, and the average latency for querying one tensor block from the index of our proposed approach to (1) Mistique exact deduplication approach, where two tensor blocks are deduplicated only if they have the same hash code; (2) Mistique approximate deduplication; and (3) Enhanced pairwise comparison approach with magnitude ordering applied. Both (2) and (3) used periodic accuracy checks, for which, we evaluate the accuracy of a model once for indexing every five blocks from the model, and we stop deduplication for a model once its accuracy drop exceeds 3.5%. However, we do not roll back to ensure the accuracy drop is within 3.5% for these experiments, though such rollbacks can be easily implemented. As illustrated in Tab. 5, the proposed approach based on L2 LSH still achieved the best compression ratio. The Mistique's approximate approach [68] is significantly slower in querying the index because a new block requires to be discretized and the MinHash generation requires multiple rounds of permutations. Due to such overhead, the latency required for building an index using the Mistique approximate approach is significantly higher than our proposed approach.

Table 5: Comparison of compression ratio and index query time.

	Blocks w/o dedup	Blocks w/ dedup	Query Time (Per Block, second)
Mistique Exact Dedup	2545	2040	0.02
Mistique Approximate Dedup	2545	712	10+
Enhanced Pairwise	2545	693	2.9
Proposed (w/o finetune)	2545	662	0.2

7.3.1 Validation Overheads Analysis. We first evaluate the storage costs of validation datasets and the additional latency incurred by the periodic accuracy validation process. As illustrated in Fig. 13, for several large-scale models, the size of an effective validation dataset is significantly smaller than the size of storage space that can be saved through deduplication.

In addition, we also implemented a *variant* of our approach that does not rely on validation datasets, but relies on the tuning of the LSH collision threshold. We split an LSH signature into multiple bands, the threshold determines the minimum band collisions required for claiming that two LSH signatures match [37, 79]. We find that by tuning the threshold, the users can achieve different

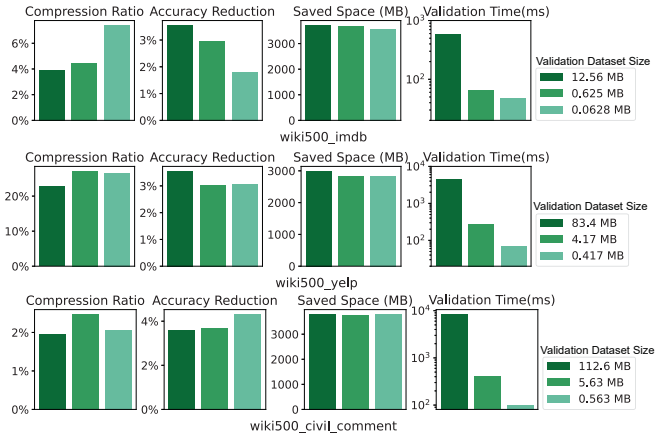

Figure 13: The compression ratio, accuracy reduction, saved space, and per-iteration validation latency for deduplicating three text classification models with different sizes of validation datasets.

levels of trade-offs between accuracy and compression ratio. This *variant* provides an alternative for applications that do not have validation datasets.

Table 6: Tuning of an LSH Threshold in the same scenario of Fig. 13 (the total number of bands is 90**).**

LSH Threshold	Compression Ratio	Accuracy Change
6	11.74%	-7.45%
7	54.34%	-0.57%
8	88.07%	0.05%

7.4 Evaluation of Page Packing Algorithms

We evaluated our proposed page packing algorithms using four evaluation scenarios: (1) Two-stage algorithm, which used Alg. 2 in stage 1, and then apply Alg. 3 to items in non-full bins in stage 2. (2) Greedy-1 algorithm that is based on equivalent classes (Alg. 2); (3) Greedy-2 algorithm that applies Alg. 3 to overall page packing. (4) DedupBase, which first packs tensor blocks to pages in order, and then eliminate the duplicate pages.

Table 7: Comparison of required number of pages using different page packing algorithms.

Scenario (block size, page size)	DedupBase	Two-Stage	Greedy-1	Greedy-2
word2vec (100 × 10000, 64MB)	130	**98**	99	**98**
text classification (100 × 10000, 64MB)	101	**87**	91	**87**
text classification (300 × 300, 64MB)	156	**104**	108	109
text classification (300 × 300, 32MB)	270	**195**	198	202
Hetero-Scenario-1(50 × 10000, 64MB)	58	55	**53**	56

We observed significant improvement in storage efficiency brought by the two-stage algorithm compared to alternatives, as illustrated in Tab. 7, except for the Heterogeneous-Scenario-1, in which pages packed in the second stage cannot be reused. In addition, the computation efficiency of the two-stage algorithm is comparable to Greedy-1, and both are below 0.1 seconds in most of the scenarios. When only applying Greedy-2, the time required to frequently compute subsets of packed pages to form a maximal subset of a tensor becomes the bottleneck, which will take 10 to 40 seconds to pack pages for the two text classification scenario with 300 × 300 blocks.

7.5 Evaluation of Caching Optimization

We also compare the proposed caching optimization to a number of baselines, including LRU, MRU, as well as the locality set page replacement policy without considering the page sharing. The detailed cache hit ratio comparison for the Word2Vec embedding and text classification applications are illustrated in Fig. 14. Locality Set-M/L refers to the locality set page replacement policy [82, 83] that treats shared pages as one locality set and applies the MRU/LRU to this locality set of shared pages. Optimized M/L refers to the localitySet-M/L with the proposed caching optimization applied (i.e., shared pages will be given a higher priority to be kept in memory). We observed that, after deduplication, the cache hit ratio improved significantly because of the reduction in memory footprint. In addition, with the proposed deduplication approach applied, Optimized-M/L achieved a significantly better cache hit ratio than alternative page replacement policies.

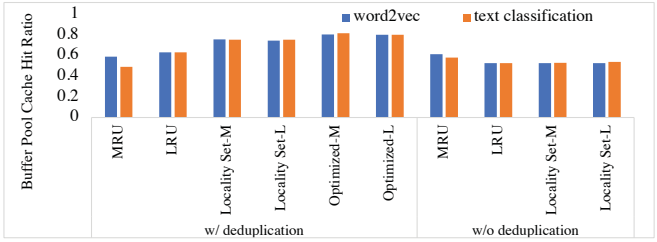

Figure 14: Comparison of different page replacement policies

7.6 Further Discussions

7.6.1 Model Updates. Deep learning models may be updated from time to time at the serving stage. We implemented and compared two approaches to deduplicate updated models.

Approach-1. The updates to a tensor are implemented as a removal of the old tensor followed by an insertion of the new tensor. To remove a tensor, all private pages belonging to the tensor will be removed, and then, for each shared page belonging to this tensor, its reference count will be decremented. Once a shared page's reference count is dropped to 1, this shared page will be moved from the shared page set to the private set of the tensor that owns the page. At the same time, the identifiers of the blocks of the tensor are also removed from the index. If a tensor block in a model needs to be removed, the LSH signature of the new block is computed to query the corresponding group for this block, and the block's identifier will be removed from the group. Adding or removing blocks to/from the group will not affect the representative block of the group. If the representative block is the only block in the group, and it is to be removed, the group will be removed.

Approach-2. We can also leverage the index to facilitate model updates at a fine-grained level. First, the LSH signature for each block in the updated model will be computed, and only the pages that involve the blocks, of which the LSH signatures have changed, need to be repacked. As illustrated in Tab. 8, we observed that both approaches achieve a similar compression ratio with limited accuracy drop. But Approach-2 is more efficient because it skips the processing (e.g., accuracy validation) of blocks that have unchanged LSH signatures.

Table 8: Deduplicating updated wiki500_imdbm model.

	compression	accuracy	validation	end-to-end duplicate detection
Approach-1	8.85%	−4.07%	44 secs	148 secs
Approach-2	10.41%	−3.61%	9 secs	108 secs

7.6.2 Relationship to Model Compression. Besides deduplication, there exist a number of model compression techniques, such as pruning [32, 33] and quantization [38], which can only be applied to each single model separately. In this work, we found that as a cross-model compression technique, model deduplication can be applied after pruning or quantizing, which achieved 2× to 3× better storage efficiency. That's because pruning and quantization will not significantly change the similarity of tensor blocks across models.

Table 9: Comparison of compression techniques (Compression ratio is defined as the ratio of the size after compression to the size before compression. Accuracy drop is measured as the maximum accuracy drop of the models after compression.)

	pruning	quantization	dedup	dedup+ pruning	dedup+quant
auc drop	3.2%	1.33%	3.98%	3.6%	3.78%
compression ratio	23.4%	12.5%	27.32%	6.74%	5.24%

8 CONCLUSIONS AND FUTURE WORKS

Serving deep learning models from RDBMS will benefit from the RDBMS' physical data independence and manageability. This work proposed synergistic storage optimization techniques covering indexing, page packing, and caching, which are implemented in netsDB, an object-oriented relational database. We evaluated these proposed techniques using several typical model serving scenarios, including the serving of (1) multiple fine-tuned word embedding models, (2) multiple text classification models, (3) multiple extreme classification models based on FFNN, and (4) multiple heterogeneous models. The results showed that our proposed deduplication techniques achieved 2.7× to 3.6× reduction in storage size, speeded up the inference by 1.1× to 4.7×, and improved the cache hit ratio by up to 1.6×. The results also showed that significantly more models can be served from RDBMS than TensorFlow, which helps to reduce the operational costs of model inferences.

We also observed that the relational processing involves additional overheads such as building join hashmap, fork-join scheduling, query optimization, and compilation. Therefore, RDBMS is mostly suitable when the models are too large to fit in memory and/or the input features are large in size and the overheads for transmitting input features from RDBMS to deep learning frameworks are unacceptable. When models can all fit in memory, and the relational processing overhead cannot outweigh the benefit brought by RDBMS, we suggest not to use our solution for applications that have stringent latency requirements. That said, aforementioned relational processing overheads can be alleviated by applying join hashmap materialization, asynchronous scheduling, and ahead-of-time query compilation, which we will study in our future works.

ACKNOWLEDGMENTS

This work was supported by ASU FSE start-up funding, IBM Academic Research Award, and NSF CAREER award (Number 2144923). We also appreciate the constructive feedbacks from the anonymous reviewers of VLDB 2022.

REFERENCES

[1] [n.d.]. The Extreme Classification Repository: Multi-label Datasets & Code. http://manikvarma.org/downloads/XC/XMLRepository.html.

[2] [n.d.]. NNLM128, Tensorflow Hub. "https://tfhub.dev/google/nnlm-en-dim128/2".

[3] [n.d.]. NNLM50, Tensorflow Hub. "https://tfhub.dev/google/nnlm-en-dim50/2".

[4] [n.d.]. shakespeare.txt. 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt'

[5] [n.d.]. Tensorflow Hub. "https://www.tensorflow.org/hub".

[6] [n.d.]. TensorFlow Wikipedia Dataset. https://www.tensorflow.org/datasets/catalog/wikipedia.

[7] [n.d.]. The total cost of ownership (tco) of amazon sagemaker. ([n. d.]). https://pages.awscloud.com/NAMER-ln-GC-400-machine-learning-sagemaker-tco-learn-ty.html.

[8] [n.d.]. Web Text Corpus. 'https://www.kaggle.com/nltkdata/web-text-corpus'

[9] [n.d.]. Wiki250, Tensorflow Hub. "https://tfhub.dev/google/Wiki-words-250/2".

[10] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. 2002. Eliminating fuzzy duplicates in data warehouses. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 586–597.

[11] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in Neural Information Processing Systems* 13 (2000).

[12] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. 2009. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, 1–9.

[13] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. 2006. Adaptive blocking: Learning to scale up record linkage. In *Sixth International Conference on Data Mining (ICDM'06)*. IEEE, 87–96.

[14] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[15] Daniel Borkan, Lucas Dixon, Jeffrey Sorensen, Nithum Thain, and Lucy Vasserman. 2019. Civil Comments Dataset. https://www.kaggle.com/c/jigsaw-unintended-bias-in-toxicity-classification/data

[16] Andrew Borthwick, Stephen Ash, Bin Pang, Shehzad Qureshi, and Timothy Jones. 2020. Scalable Blocking for Very Large Databases. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 303–319.

[17] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 21–29.

[18] Ilias Chalkidis, Manos Fergadiotis, Prodromos Malakasiotis, and Ion Androutsopoulos. 2019. Large-scale multi-label text classification on EU legislation. *arXiv preprint arXiv:1906.02192* (2019).

[19] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[20] Lin Chen, Hossein Esfandiari, Gang Fu, and Vahab Mirrokni. 2019. Locality-Sensitive Hashing for f-Divergences: Mutual Information Loss and Beyond. In *Advances in Neural Information Processing Systems*. 10044–10054.

[21] Hong-Tai Chou and David J. DeWitt. 1986. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica* 1, 3 (1986), 311–336. https://doi.org/10.1007/BF01840450

[22] Xu Chu, Ihab F Ilyas, and Paraschos Koutris. 2016. Distributed data deduplication. *Proceedings of the VLDB Endowment* 9, 11 (2016), 864–875.

[23] Daniel Crankshaw, Xin Wang, Joseph E Gonzalez, and Michael J Franklin. 2015. Scalable training and serving of personalized models. In *NIPS 2015 Workshop on Machine Learning Systems (LearningSys)*.

[24] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[25] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[26] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory.. In *USENIX annual technical conference*. 1–16.

[27] Oksana Dolmatova, Nikolaus Augsten, and Michael H Böhlen. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2573–2587.

[28] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. 2006. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering* 19, 1 (2006), 1–16.

[29] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.

[30] James F Gimpel. 1974. The minimization of spatially-multiplexed character sets. *Commun. ACM* 17, 6 (1974), 315–318.

[31] Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* (2014).

[32] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[33] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626* (2015).

[34] Mauricio A Hernández and Salvatore J Stolfo. 1995. The merge/purge problem for large databases. *ACM Sigmod Record* 24, 2 (1995), 127–138.

[35] Daniel P. Heyman. 1977. Queueing Systems, Volume 2: Computer applications. by Leonard Kleinrock John Wiley & Sons, Inc., New York 1976, 549 Pages, $24.95. *Networks* 7, 3 (1977), 285–286. https://doi.org/10.1002/net.3230070308

[36] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM, 2.

[37] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[38] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

[39] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2019. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *Proceedings of the VLDB Endowment* 12, 7 (2019), 822–835.

[40] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2019. Extending relational query processing with ML inference. *arXiv preprint arXiv:1911.00231* (2019).

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

[42] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load balancing for mapreduce-based entity resolution. In *2012 IEEE 28th international conference on data engineering*. IEEE, 618–629.

[43] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804. http://www.vldb.org/pvldb/vol14/p1797-koutsoukos.pdf

[44] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 175–190.

[45] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. {PRETZEL}: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 611–626.

[46] David D Lewis, Yiming Yang, Tony Russell-Rose, and Fan Li. 2004. Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research* 5, Apr (2004), 361–397.

[47] Peipei Li, Xindong Wu, and Xuegang Hu. 2010. Mining recurring concept drifts with limited labeled streaming data. In *Proceedings of 2nd Asian conference on machine learning*. JMLR Workshop and Conference Proceedings, 241–252.

[48] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. 2016. CacheDedup: In-line deduplication for flash caching. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 301–314.

[49] Jianhua Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information theory* 37, 1 (1991), 145–151.

[50] Xin Liu and Kenneth Salem. 2013. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment* 6, 8 (2013), 541–552.

[51] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2018), 1224–1238.

[52] Alexander Ly, Maarten Marsman, Josine Verhagen, Raoul PPP Grasman, and Eric-Jan Wagenmakers. 2017. A tutorial on Fisher information. *Journal of Mathematical Psychology* 80 (2017), 40–55.

[53] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Large Movie Review Dataset. http://ai.stanford.edu/~amaas/data/sentiment/

[54] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. 165–172.

[55] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 785–794.

[56] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 43–52.

[57] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[58] Dutch T Meyer and William J Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage (ToS)* 7, 4 (2012), 1–20.

[59] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[60] Simon Mo, Edward Oakes, and Michael Galarnyk. [n.d.]. Serving ML Models in Production: Common Patterns. ([n. d.]).

[61] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 899–917.

[62] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[63] Dan Olteanu. 2020. The relational data borg is learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3502–3515.

[64] Haichen Shen, 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. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[65] Larry J Stockmeyer. 1975. *The set basis problem is NP-complete*. IBM Thomas J. Watson Research Division Research reports.

[66] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *International Conference on Scientific and Statistical Database Management*. Springer, 1–16.

[67] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. 2007. The role mining problem: finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies*. 175–184.

[68] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*. 1285–1300.

[69] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. 2020. Austere flash caching with deduplication and compression. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 713–726.

[70] Wei Wang, Sheng Wang, Jinyang Gao, Meihui Zhang, Gang Chen, Teck Khim Ng, and Beng Chin Ooi. 2018. Rafiki: machine learning as an analytics service system. *arXiv preprint arXiv:1804.06087* (2018).

[71] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951* (2020).

[72] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment* 1, 1 (2008), 933–944.

[73] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 2016. A generic method for accelerating LSH-based similarity join processing. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2016), 712–726.

[74] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2020. Tensor Relational Algebra for Machine Learning System Design. *arXiv preprint arXiv:2009.00524* (2020).

[75] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.

[76] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Yelp polarity Reviews Dataset. http://goo.gl/JyCnZq

[77] Lixi Zhou, Zijie Wang, Amitabh Das, and Jia Zou. 2020. It's the Best Only When It Fits You Most: Finding Related Models for Serving Based on Dynamic Locality Sensitive Hashing. *arXiv preprint arXiv:2010.09474* (2020).

[78] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system.. In *Fast*, Vol. 8. 269–282.

[79] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH ensemble: Internet-scale domain search. *arXiv preprint arXiv:1603.07410* (2016).

[80] Jia Zou, 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. In *Proceedings of the 2018 International Conference on Management of Data*. 1189–1204.

[81] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. Lachesis: Automated Partitioning for UDF-Centric Analytics. *Proc. VLDB Endow.* 14, 8 (2021), 1262–1275. https://doi.org/10.14778/3457390.3457392

[82] Jia Zou, Arun Iyengar, and Chris Jermaine. 2019. Pangea: monolithic distributed storage for data analytics. *Proceedings of the VLDB Endowment* 12, 6 (2019), 681–694.

[83] Jia Zou, Arun Iyengar, and Chris Jermaine. 2020. Architecture of a distributed storage that combines file system, memory and computation in a single layer. *The VLDB Journal* (2020), 1–25.