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 serving models from relational databases outperformed existing deep learning frameworks when the working set size exceeds available memory.

ACM Reference Format:
Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, and Jia Zou. 2022. Serving Deep Learning Models with Deduplication from Relational Databases . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [56], Clipper [22], Pretzel [42], TensorFlow Serving [54], and Rafiki [62], 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 models, input features, and intermediate feature maps all fit in memory. Failing to meet such requirements leads to the failing of the system. For large models with large working set, which is common in applications such as natural language processing and extreme multi-label classification [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) [12, 25, 31, 34, 36, 40, 53, 63, 66]. RDBMS has a long history of optimizing the memory locality for computations (i.e., queries), 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 [55], such as credit-card fraud detection, personalized targeting recommendation, and personalized 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.

As aforementioned, unlike deep learning frameworks, workloads in RDBMS are not expected to have a working set fit in the available memory. The RDBMS buffer pool manager moves pages between disk and memory to optimize the data locality while continuing query processing. This allows more models to be served concurrently than deep learning frameworks such as TensorFlow with the same memory capacity. Nonetheless, there is always a desire to *increase buffer reuse and minimize page displacement.* To achieve this in model serving, we look into **model deduplication**.

Serving multiple similar models, such as ensemble and personalized model serving, can greatly improve the accuracy and customer experiences, and thus becomes a common pattern of DNN model serving [21, 22, 52]. Such DNN models contain abundant *similar* tensor blocks that can be deduplicated without affecting the accuracy. As a result, proper deduplication of such DNN models significantly reduces the storage space, memory footprint, and cache misses, and thus reduces the inference costs and latency.

However, existing deduplication techniques for tensors [59], files [10, 24, 43, 51, 61, 70], relational data [9, 11, 14, 26, 29, 64, 65], and MapReduce platforms [19, 37, 38], are not applicable to the above problem, 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?
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. We mainly leverage our previous works on Tensor Relational Algebra [34, 66] to map deep learning computations to relational algebra expressions. 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 [12], Spark MLlib [50], SciDB [58], SPORES [63], LaraDB [31], 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 [41]. To address the problem, different from existing tensor deduplication works [59], we propose to first measure each tensor block's sensitivity to prediction accuracy based on weight magnitude or other post-hoc analysis [28], 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 [32, 69], 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 bin-packing problem with different constraints: (1) Two bins (i.e., pages) can share space if they contain the same set of items (i.e., tensor blocks) [39, 57]; (2) For each tensor, there must exist a set of pages that exactly contain all blocks of that tensor. 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 employs a divide-and-conquer approach to pack tensor blocks in each equivalent class to pages respectively, and later it adopts an approximation algorithm to repack the tensor blocks from non-full pages.

Deduplication-aware buffer pool management (Sec. 6). Existing deduplication-aware cache replacement strategies [43, 61] do not consider the locality patterns of different sets of pages, which are important for model inference workloads where the input and the output of each layer have different locality patterns. However, existing locality-aware buffer pool management policies [18] do not distinguish private pages and shared pages. To address this problem, we propose a cost model for page eviction, which considers the reference count of a page (i.e., the number of locality sets/tensors that share this page) and 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 the tensor blocks across 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 [27, 28] and quantization [33] to achieve a better compression ratio, as detailed in Sec. 7.6.
3. We implement the system in an object-oriented relational database based on our previous work of PlinyCompute [71–74], 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. 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 AND RELATED WORKS

2.1 ML Model Inferences as Queries

Existing works [12, 45, 50] 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. Based on the above ideas, tensor relational algebra (TRA) [66] further introduces a set of tensor-oriented relational operations, such as `tile`, `concat`, `rekey`, `transform`, `join`, `aggregation`, `selection`, etc. We found that most ML workloads can be decomposed into linear algebra operations that are further represented in such TRA.

For example, **matrix multiplication** is a `join` followed by `aggregation` [12, 66]. 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

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

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, **matrix transpose** is a `rekey` [66]; **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 net

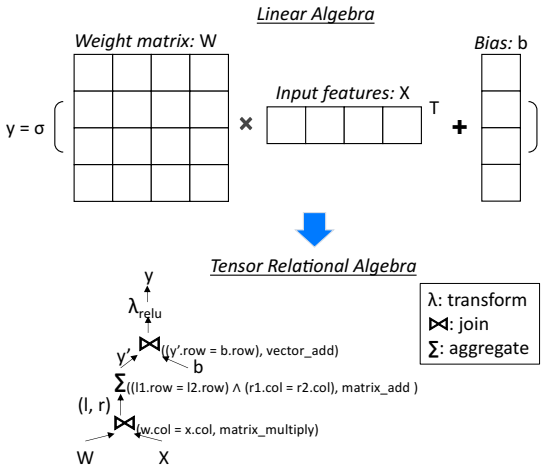

Figure 1: Example of mapping linear algebra to relational algebra.

While the experiments in this work (Sec. 7), mainly used aforementioned operators, other types of neural networks can also be represented in relational algebra. For example, convolution can be converted into a multiplication of two matrices [2, 3], where the first matrix is created by spatially flattening every filtered area of the input features into a vector, and concatenating these vectors, and the second matrix is created by concatenating all filters and bias. Long short-term memory (LSTM) consists of `concat`, matrix multiplication, matrix addition, tanh, and sigmoid; and the transformer's attention mechanism consists of matrix multiplication, transpose, softmax, etc [60].

The storage optimization techniques proposed in this work can be easily extended to other tensor/array-based machine learning systems, which adopt a similar tensor representation that chunks a tensor to blocks, such as SystemML [12], Spark MLlib [50], SciDB [58], SPORES [63], LaraDB [31], etc. In contrast, Raven [36] and HummingBird [53] propose to transform relational data to tensors and leverage deep learning frameworks to run tensor computations. We will investigate how to apply proposed deduplication techniques to such and other systems [25, 40] in the future.

2.2 Tensor Deduplication and Virtualization

Mistique [59] proposed a data store for managing and querying the historical intermediate data generated from ML models. It optimized the storage of fuzzy intermediate data using quantization, summarization, and deduplication. However, these techniques are designed for diagnosis queries, which are not linear algebra computations and have significantly less stringent accuracy and latency requirements compared to model inferences. While they considered both exact and approximate deduplication for traditional ML models, they only considered exact deduplication for DNN models, which is another limitation. In addition, they didn't consider page packing and caching optimization.

Jeong and et al. [35] proposed to merge related models resulting from ensemble learning, transfer learning, and retraining into a single model through input-weight connectors, so that multiple models can be served in one process and context switch overheads caused by running multiple concurrent model processes can be avoided. However, their method makes strong assumptions about the model architecture, achieves only coarse-grained deduplication, and is not applicable to models that are owned by different individuals and organizations.

Weight virtualization [41] is a recently proposed technique for edge device environments. It merges pages across multiple heterogeneous models into a single page that is shared by these models. However, their work relied on each weight's fisher information that must be extracted from the training process, which is usually not available at the serving stage in production. It also models the page matching and merging process as an expensive optimization process, which may work for small-scale models on edge devices, but not scalable to large-scale models. In addition, they didn't consider the integration with relational databases.

2.3 Other Existing Deduplication Techniques

Deduplication of relational data in RDBMS, also known as record linkage, identifies duplicate items through entity matching [26], using various blocking techniques to avoid the pair-wise comparison for dissimilar items [9, 11, 14, 29]. Various distributed algorithms were proposed to further accelerate such deduplication [19]. For example, Dedoop [37, 38] leveraged the MapReduce platform, and Dis-Dedup [19] provided strong theoretical guarantees for load balance. In addition, various similarity join techniques were proposed to identify pairs of similar items, which leveraged similarity functions to filter out pairs that have similarity scores below a threshold [64] or used LSH to convert similarity join to an equi-join problem [65]. While these works are helpful for cleaning data in RDBMS, they are not optimized for numerical tensor data. For example, they never considered how deduplication of tensor data will affect the accuracy of ML applications.

There exists abundant work in storage deduplication to facilitate the file backup process [51]. Bhagwat et al. [10] proposed a two-tier index managing the fingerprints and file chunks. Zhu et al. [70] proposed RAM prefetching and bloom-filter based techniques, which can avoid disk I/Os on close to 99% of the index lookups. ChunkStash [24] proposed to construct the chunk index using flash memory. CacheDedup [43] proposed duplication-aware cache replacement algorithms (D-LRU, DARC) to optimize both cache performance and endurance. AustereCache [61] proposed a new flash caching design that aims for memory-efficient indexing for deduplication and compression. All such works focus on exact deduplication of file chunks, because information integrity is required for file storage. However, the storage of model parameters

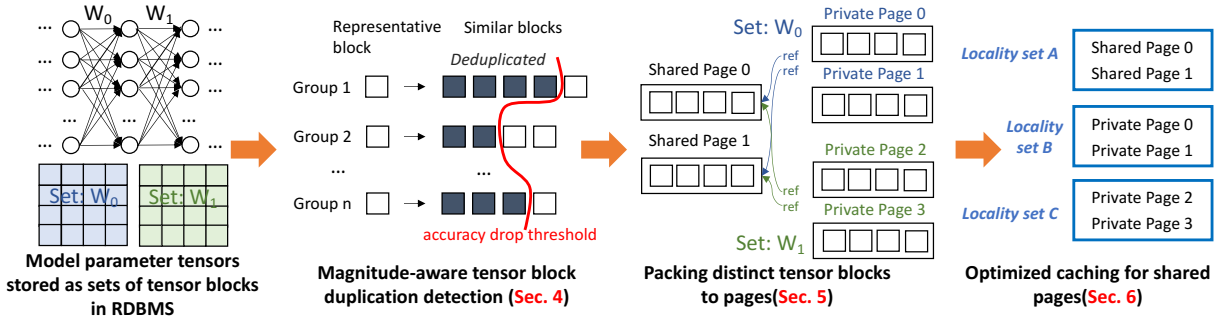

Figure 2: Overview of the proposed model deduplication workflow.

for model serving can tolerate a certain degree of approximation if such approximation will not harm the inference accuracy.

3 SYSTEM OVERVIEW

Leveraging tensor relational algebra [34, 66], a tensor is represented as a set of tensor blocks[2]. 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. 2 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 examining 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)

(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 through a novel cost model. (Sec. 6)

Block Metadata. A major portion of overhead of the proposed deduplication mechanism is incurred by the additional metadata used to map each tensor block in these shared pages to the correct position in each tensor. Each tensor block needs $m \times d$ integers to specify such mapping, where m is the number of tensors that share the block and d is the number of dimensions of the tensor. The metadata size is usually much smaller than the block size. For an 8 megabytes block (e.g., 100×10000 with double precision), its metadata for position mapping is merely 400 bytes, supposing such a 2D block is shared by 100 tensors, using short type to store block indexes. Even when we use small block sizes such as 100×100, the block size is hundreds times larger than the metadata size.

As aforementioned, an important pattern of model serving involves multiple versions of models that have the same architecture, e.g., obtained by retraining/finetuning a model using different datasets. We found that the deduplication of such models does not require tensor block remapping at all, as a shared tensor block is often mapped to the same position of all tensors it belongs to. That's because during the process of finetuning and retraining, only partial weights will change. For a tensor block in such scenarios, we only need m integers to specify the IDs of tensors that share it.

Model Removal and Updates. 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. Given that the models in a serving scenario are less frequently updated than models in a training scenario, an update is implemented as a removal of the old tensor followed by an insertion of the new tensor. However, the index can be easily extended to facilitate model updates at a fine-grained level, as discussed in Sec. 4.

4 INDEX FOR DUPLICATION DETECTION

4.1 Problem Description

In this section, we focus on one problem: *For the tensors with same blocking shapes, how to divide all tensor blocks of these tensors into distinct groups, so that the tensor blocks in each group can replace each other without a significant drop in the inference accuracy of each model?* We can further pick one block, i.e., the first identified block, in each group as a representative tensor block to replace other blocks in its group, without significant accuracy drop. The problem is formalized as follows:

[2]Luo et al [46] proposed an auto tuning strategy for blocking tensors for TRA [66].

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.

4.2 Main Ideas

4.2.1 Magnitude-aware Duplicate Detection. Existing works about deduplication [9, 11, 14, 19, 26, 29, 37, 38, 43] and tensor chunk deduplication [59], include exact page deduplication and similar/approximate page deduplication, as detailed in Sec. 2.2 and 2.3. 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. In fact, the number of tensor blocks that can be deduplicated based on exact match is significantly lower than similarity-based match.

(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 [27, 28], 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 [23], Euclidean distance [32], and cosine similarity [16] 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 [15] is designed to identify similar binary vectors or similar sets of items. In this work, we mainly use the LSH based on Euclidean distance [17, 32], 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 [44] of weights' probability distributions of two tensor blocks [17].

For each block, its LSH signature is computed and used as the search key, and the identifier of the block (`TensorID`, `BlockID`) is used as the value. The key-value pair is sent to an index to look up a group of similar blocks that collide on the signature. For each group, the *first indexed block* is used as the representative block of this group, and other blocks are replaced by this representative block if accuracy drop is tolerable. If another block in the group has the same `BlockID` with the representative block, the `BlockID` field, which encodes the block's position along all dimensions of the tensor, can be omitted to save space.

4.3 Index Building

Given a set of models, we execute following steps *for each model*:
Step 1. Calculate an aggregated magnitude value (e.g., average, median, 1st percentile, 3rd percentile, etc.) for each tensor block in the tensors of the model. We use the 3rd percentile, because even if the block contains only a few large magnitude weights, it may impact the inference accuracy significantly and should not be deduplicated. 3rd percentile can better reflect the magnitude of large weights in this block than aforementioned alternatives.
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 and used to query the index. If the index has seen similar blocks before, the block's identifier will be added to the corresponding group and this block will be replaced by the representative block, which is the first indexed block in this group. If the index hasn't seen similar block before, a new group will be created, and this block becomes the representative block in the group.
Step 4. We will 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 [3], each of which has only one physical copy.

We repeat the above process for each model to incrementally construct the index, as illustrated in Alg. 1. The inputs of the algorithm include: (1) $T = \{t_1, ..., t_k\}$, which is a set of tensors belonging to the model; (2) idx, which maps an LSH signature to a representative block d_c and a cluster c consisting of the identifiers of blocks of which the signatures collide and thus are similar to the representative block; (3) L, which is a list of distinct tensor blocks derived from previous models. The idx and L are *shared by all models* and will be updated during the execution of the algorithm.

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 information is needed to pack distinct tensor blocks to pages as detailed in Sec. 5.

Further Optimizations. 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.

Removal and Updates. If a tensor block in a model needs to be removed, the LSH signature of the block is computed to query the index. If there exists a match and the block's identifier exists in the corresponding group, the identifier will be removed from the group. Adding or removing blocks from the group will not affect

[3] It is possible a remaining block is also a representative block in its own group.

Algorithm 1 Index Building

INPUT1: $T = \{t_1, ..., t_k\}$ (A set of parameter tensors in a model)
INPUT2: idx (The index that has been constructed for previous models, and will be updated by this model.)
INPUT3: $L = \{d_1, ..., d_m\}$ (A set of distinct blocks derived from previous models, which will be updated by this model.)
OUTPUT: $F_T = \{f_1, ..., f_k\}$ (f_i maps each tensor block in t_i to a distinct block)
$B = \{b_1, ..., b_n\} \leftarrow \cup_{i=1}^{k} t_i$
$a_0 \leftarrow accuracy(Model_B)$
for $i = 1, ...n$ **do**
  $(b_i, v_i) \leftarrow (b_i, getMagnitude(b_i))$
end for
$B' = \{b'_1, ..., b'_n\} \leftarrow$ sort B by v_i in ascending order
$i \leftarrow 0$
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

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. The update of a tensor block can be regarded as a removal followed by an insertion.

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.

5.1 Inconsistent Pages and Tensor Blocks

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 [50]). 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 are packed into one page. Then a page deduplication process is performed, so that each distinct page will be physically stored once. However, such default packing with page-level deduplication is sub-optimal, because deduplicable tensor blocks may not be adjacent to each other spatially. As illustrated in Fig. 3, the default packing requires 8 pages, while the optimal packing scheme requires only 5 pages.

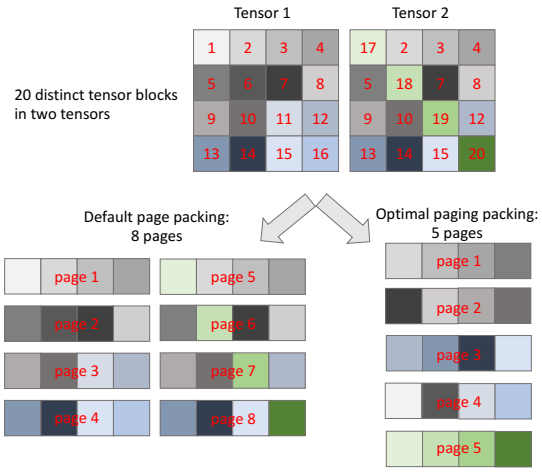

Figure 3: Motivation of page packing optimization

5.2 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 problem definition as a variant of the bin packing problem, where each **bin** represents a page that holds a limited number of tensor blocks, and each distinct tensor block represents an **item**. Given k tensors $T = \{t_1, ..., t_k\}$ and a set of distinct tensor blocks $I = \{item_1, ..., item_m\}$ derived from these tensors, a Boolean value a_{ij} specifies whether $item_i$ exists in the j-th tensor. $\forall t_i \in T$, $t_i \subset I$, as described in Sec. 4. The problem is to look for a bin-packing scheme that packs the items (i.e., distinct tensor blocks) to n bins (i.e., pages), denoted as $bins = \{bin_1, ..., bin_n\}$, where each bin can hold at most l items and each item can be allocated to one

or more bins, denoted as $bin_i \subset I$ and $|bin_i| \leq l$. Boolean value p_{ij} denotes whether $item_i$ exists in bin_j. The bin packing mechanism $P = \{p_{ij}\}$ must satisfy conditions as follows: (1) the total number of bins, $\sum_{j=0}^{n} y_j$, is minimized, where the Boolean value denotes whether the bin_j is used; (2) $\forall t_i \in T, \exists bins' \subset bins$, so th $t_i = \cup_{bin \in bins'} bin$, which means the set of distinct items contain in a tensor t_i is equivalent to the set of distinct items contained all bins belonging to $bins'$.

$$\min \sum_{j=0}^{n} y_j \quad (1)$$

$$y_j = \begin{cases} 1 & if \sum_{i=0}^{m} p_{ij} > 0 \\ 0 & otherwise \end{cases} \quad (2)$$

$$\forall bin_j \in bins, bin_j \subset I, p_{ij} = \begin{cases} 1 & if item_i \in bin_j \\ 0 & otherwise \end{cases} \quad (3)$$

$$s.t. \quad \forall j, \sum_{i=0}^{m} p_{ij} \leq l \quad (4)$$

$$\forall t_j \in T, t_j = \{item_k | a_{kj} = 1\}, \exists bins' \subset bins, t_j = \cup_{bin \in bins'} bi \quad (5)$$

Problem Importance and Hardness. It is an important proble because large page sizes up to hundreds of megabytes, are wide adopted in analytics databases [67] and when memory resource becomes insufficient, even saving only a few pages may significantly reduce the memory footprint and improve the performance.

The problem is a variant of the bin-packing problem where items (i.e., distinct blocks) can share space when packed into a bin (i.e., pages) [39, 57], which is NP-hard. A dynamic programming strategy, which searches packing plans for one tensor first, and then repeatedly pack for more tensors based on previously searched packing plans, will easily fail with exploded search space.

5.3 Equivalent Class-based Packing

While approximation algorithms [20], such as Best-fit, First-fi Next-fit, are widely used for general bin-packing problems, the are suboptimal for the above problem, because they didn't conside how tensor blocks are shared by tensors.

To solve the problem, we propose to group tensor blocks that ar shared by the same set of tensors together into **equivalent classe** Different tensor blocks that are shared by the same set of tensor are regarded as equivalent in terms of page packing.

As illustrated in Fig. 4, which depicts the tensor sharing rel tionship for the example in Fig. 3, 12 distinct blocks are shared b Tensor 1(t_1) and Tensor 2(t_2), these distinct tensor blocks can b grouped to the same equivalent class C_3. Four distinct tensor block are private to t_1 and they can be grouped to the same equivalen class C_1, and so do the blocks private to t_2 (C_2).

It is beneficial to use a divide and conquer strategy to pack fc each equivalent class in parallel by grouping the blocks fallin into the same equivalent class to the same page(s). That's becaus each page can be shared by all tensors associated with the page's corresponding equivalent class. By doing so, in the above example (Fig. 3 and Fig. 4), the 12 distinct blocks in equivalent class C_3 will be packed to three pages, the four distinct blocks in C_1 will be packed to one page, and the four distinct blocks in C_2 will be packed to one page, which leads to the optimal plan, as shown in Fig. 3. The algorithm is illustrated in Alg. 2.

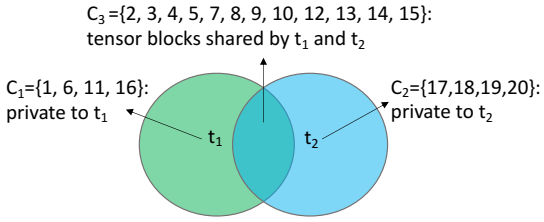

Figure 4: Illustration of equivalent classes of tensor blocks for page packing for the example in Fig. 3.

Algorithm 2 Equivalent Class-Based Greedy Strategy

```
INPUT1: T (a list of tensors)
INPUT2: l (the maximum number of items for each bin)
OUTPUT: P = {p_ij} (a bin-packing scheme)
{C_1, ..., C_m} ← T {divide I into multiple equivalent classes, so items
  in each class are shared by the same set of tensors}
numBins ← 0
for k=0..m do
  for item : C_k do
    i ← indexInI(item)
    j ← numBins + ⌈indexInC_k(item)/l⌉
    p_ij ← 1
  end for
  numBins ← numBins + ⌈|C_k|/l⌉
end for
return P = {p_ij}
```

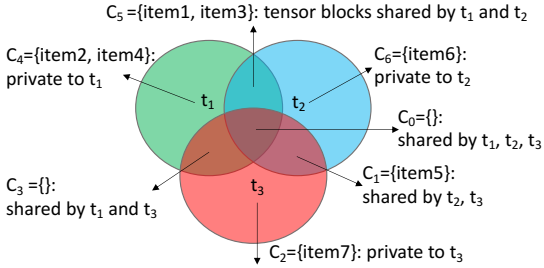

Figure 5: Another example: the equivalent class-based greedy strategy leads to three non-full pages for C_1, C_2, C_6.

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 bins. For another example as illustrated in Fig. 5, if a bin can maximally hold two items, the items in C_1, C_2, C_6 will be packed to three non-full bins respectively. However, a better scheme is to pack these items into two bins: $bin_1 = C_1 \cup C_6$ and $bin_2 = C_1 \cup C_2$. Considering that a page may have a size up to tens or hundreds of megabytes, and repacking non-full pages will enable significant improvement in storage efficiency, memory footprint, and data locality. Therefore, we propose a two-stage

strategy for optimizing page packing schemes. At the first stage, items from each equivalent class are packed to bins separately, and no bin is allowed to mix items from different non-equivalent classes. Then, at the second stage, we repack items from non-full bins, by 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 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 based on their sharing frequency (i.e., the number of tensors a block is shared by), and simply packs these blocks to 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.

Algorithm 3 Approximation Strategy

INPUT1: $T = \{t_1, ..., t_k\}$ (A set of tensors for packing to pages. When applied to Stage-2, each tensor only contains items from non-full bins resulted from Stage 1)
INPUT2: l (the maximum number of items for each bin)
OUTPUT: $P = \{p_{ij}\}$ (a bin-packing scheme)
$T \leftarrow orderByNumTensorBlocksDescend(T)$
$I \leftarrow \phi$
while $t_i \in T$ **do**
  $I \leftarrow I \cup t_i$
end while
$numBins \leftarrow 0$
for $i = 1, ..., k$ **do**
  if i > 1 **then**
    $bins \leftarrow$ a set of existing bins that form a maximal subset of t_i
    $I_\delta \leftarrow t_i - \cup_{bin \in bins} bin$
    if $I_\delta = \phi$ **then**
      continue
    end if
  else
    $I_\delta \leftarrow t_1$
  end if
  $\{item_1, ..., item_{\delta i}\} \leftarrow orderBySharingFreqDescend(I_\delta)$
  for $j = 1, ..., \delta i$ **do**
    $s \leftarrow indexInI(item_j)$ //index of $item_j$ in I
    $u \leftarrow numBins + \lceil \delta i / l \rceil$
    $p_{su} \leftarrow 1$
  end for
  $numBins \leftarrow numBins + \lceil \delta i / l \rceil$
end for
return $P = \{p_{ij}\}$

Online Packing The proposed algorithms can also be utilized for online packing of tensor blocks to pages. Each time when a new tensor is about to be added to the database, the list of tensor blocks in this tensor as well as all related tensors (i.e., tensors which share at least one block with the new tensor) will be retrieved to run the proposed algorithm to obtain a new packing scheme. Then the difference between the new packing scheme and the existing packing scheme will be computed. Only these pages that need to be changed will be repacked again.

6 BUFFER POOL MANAGEMENT

A model serving workload involves multiple types of tensors that have different locality patterns. For example, the model parameter tensors at each layer are persisted to disk and are repeatedly read for making inferences; the input feature vector also needs to be persisted, but is read only once. The intermediate features output from each layer do not need to be persisted and are read only once.

Existing works proved that compared to LRU/MRU/LFU, which only consider reference time/distance/frequency, a fine-grained buffer pool management strategy that groups different types of data based on a locality set abstraction [18, 73, 74] and considers the access pattern and durability requirements of each locality set, can achieve better data locality for large-scale data analytics processing [73, 74]. A locality set is a set of pages that will be processed similarly. For example, the pages in each equivalent class are regarded as a separate locality set. Users can configure the page eviction policy, e.g., MRU or LRU, for each locality set. 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 locality set if the next page-to-be-evicted from the locality set has the lowest expected eviction cost among all locality sets. The expected eviction cost is formalized in Eq. 6.

$$c_w + p_{reuse} \times c_r \tag{6}$$

Here, c_w is the cost for writing out the page, c_r is the cost for loading it back for reading, and p_{reuse} is the probability of accessing the page within the next t time ticks. The formulation of c_w and c_r in existing works [18, 73, 74] have considered the lifetime, durability requirements, access patterns, etc. of each locality set, and can be reused for this work. However, when modeling p_{reuse}, existing works did not consider page sharing caused by model deduplication. To address the problem, we need to reformulate this factor.

In the scenario of serving multiple models, we propose to apply the queueing theory [30] 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 [30], the probability of reusing each page (i.e., the probability that the page will be accessed within t time ticks) can be estimated using Eq. 7. Here, $M = \{m_1, ..., m_s\}$ represents a set of models that share this page, and λ_i denotes the access rate per time tick for the model m_i.

$$p_{reuse} = 1 - e^{-\sum_{m_i \in M} \lambda_i t} \tag{7}$$

This approach is more accurate than simply estimating p_{reuse} based on the reference 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.

7 EVALUATION

In this section, we will answer the following questions:

(1) How effective is the proposed synergistic model deduplication mechanism in reducing the latency and improving the storage efficiency for various model serving scenarios? (Sec. 7.2)
(2) How will the proposed index approach affect the time required for detecting the duplicate blocks, the overall storage efficiency, and the model serving accuracy? (Sec. 7.3)
(3) How will the proposed strategies of packing blocks to pages affect the storage efficiency and the computation overheads, compared to various baselines? (Sec. 7.4)
(4) How will optimized caching improve memory locality? (Sec. 7.5)
(5) How will deduplication work with popular model compression techniques, such as pruning and quantization? (Sec. 7.6)

7.1 Evaluation and Workloads

7.1.1 Multiple Versions of Personalized Text Embedding Models. Text embedding is important for many natural language processing applications, and its accuracy can be greatly improved using 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 a separate model on both the shared open data and the private domain/enterprise data. Word2Vec is a two-layer neural network used to generate word embeddings. We use skip-gram Word2Vec as well as negative sampling, with 64 negative samples, and noise contrastive estimation (NCE) loss. We deploy a Word2Vec model pretrained using a Wikipedia dump and downloaded from TFHub [5]. The model embeds the 1 million most frequent tokens in the corpus. 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 [68], IMDB reviews [47]. The input document is processed with a skip window size of 1. The Word2Vec embedding layer has one million 500 dimensional embedding vectors corresponding to one million words in the dictionary. Therefore, a weight tensor is in the shape of $10^6 \times 500$.

The inference of a word2vec model on netsDB is implemented via matrix multiplication, where an input feature vector is of the shape of $[100, 10^6]$, representing a batch of 100 input words, sentences, or documents. A word can be represented as a 10^6 dimensional one-hot encoding vector, where the corresponding word in the vocabulary is specified as 1, and other words are specified as 0. Then, multiplying the batch of encoding vectors of words with the embedding weight matrix will output the batch of embedding vectors for these words. Similarly, the encoding vector for a sentence or a document, which is seen as a "bag of words", can be represented as the sum of the one-hot encoding vectors of all the words in this sentence or document. By multiplying the batch of encoding vectors and the embedding weight matrix, the embedding for each sentence or document is obtained as the weighted sum of the embedding vectors of the words in this sentence or document.

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 [13, 47, 68]. All tasks use the same model architecture. Each model uses three layers. The first layer is a Word2Vec layer as mentioned in Sec. 7.2.1, using a vocabulary size of one million 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 layer is small in size, we encode it in a UDF that is applied to the output of the Word2Vec embedding layer.

The first two text semantic classification models are trained using the same IMDB review datasets. The difference is that Model-1's Word2Vec 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 Yelp datasets, with the Word2Vec layer set to be `Non-Trainable` and `Trainable` respectively. The Model-5 is trained using the civil comments [13], which are collected from news sites with labeled toxicity values, and the Word2Vec layer in this model is set to be `Trainable`.

7.1.3 Transfer Learning of Extreme Classification Models. Following TRA [66], a two-layer feed-forward neural network (FFNN) is implemented in our proposed system for the AmazonCat-14K [48, 49] benchmark. 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 $597,540,000$ parameters, and the weight tensor (denoted as W_2) in the second layer has $14,588,000$ 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.

The implementation of the feed-forward inference at each fully-connected layer is illustrated in Fig. 1.

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.

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 Word2Vec model after finetuning is above 99%.) Each model is a $1,000,000 \times 500$ tensor, stored in a set of tensor

blocks in the shape of $10,000 \times 100$, each weight is stored in double precision. 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 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.

In Tab. 1 and 2, 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 Tab. 1 and Tab. 2.

Table 1: 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. (Unit: seconds)

num models	disk type	w/o dedup	w/ dedup & optimized caching
2	SSD	191	175
3	SSD	350	262
4	SSD	506	381
6	SSD	720	513
2	HDD	430	425
3	HDD	1112	639
4	HDD	1474	962
6	HDD	2209	1398

Table 2: Overall latency for serving six word2vec models using different storage configurations (Unit: Seconds)

disk type	buffer pool size	w/o dedup	w/ dedup	w/ dedup & optimized caching
SSD	15GB	720	513	513
SSD	10GB	762	594	580
SSD	8GB	786	710	638
HDD	15GB	2209	1398	1398
HDD	10GB	2264	1435	1435
HDD	8GB	8120	4921	1720

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`), which is similar to the netsDB implementation of Word2Vec inference as mentioned in Sec. 7.1.1. In the experiments of comparing this approach and netsDB, we used double precision for both systems.

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. For netsDB's implementation based on matrix multiplication, 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, as mentioned in Sec. 7.1.1. 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. 3, TF-mem, TF-file, and TF-DB loads 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 a batch of such inferences cannot fully utilize the GPU parallelism.

Table 3: 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

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 deduplication as well as the classifier inference accuracy before and after deduplication. The comparison results are illustrated in Tab. 4.

Without deduplication, the total storage space required is 20.5GB for 320 pages in total. After applying the proposed deduplication mechanism, the total storage space required is reduced to 5.6GB for 87 pages, using the block size of 100×10000.

Table 4: Pages deduplicated (shared pages) and inference accuracy before and after deduplication. (Unit: Seconds)

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

Table 5: Page reference count distribution after deduplication

	Model-1	Model-2	Model-3	Model-4	Model-5	Total
pages shared by 5 models	51	51	51	51	51	51
pages shared by 4 models	6	6	6	0	6	6
pages shared by 3 models	5	0	5	0	5	5
pages shared by 2 models	0	0	1	0	1	1
private pages	2	7	1	13	1	24

Each shared page may have a different reference count (i.e., shared by a different set of tensors). So we illustrate the reference counts of pages for each model in Tab. 5.

The comparison of the overall inference latency of all five text classification models, using different block sizes and storage configurations, is illustrated in Tab. 6. 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 ($40\% - 46\%$). Because this is a transfer learning scenario, the shared pages have no approximation at all, there exists no influence on accuracy.

Table 6: Overall latency for serving text classification models using different storage configurations. (Unit: Seconds)

disk type	buffer pool size	w/o dedup	w/ dedup	w/ dedup & optimized caching
SSD	15GB	646	427	426
SSD	10GB	655	572	540
SSD	8GB	675	595	557
HDD	15GB	1,675	1,091	1,085
HDD	10GB	1,815	1,515	1,467
HDD	8GB	1,815	1,686	1,620

Table 7: Overall deduplication results for transfer learning with FFNN. (Unit: Seconds)

disk type	buffer pool size	w/o dedup	w/ dedup	w/ dedup & optimized caching
SSD	9GB	115	109	103
SSD	13GB	114	96	96
HDD	9GB	221	203	157
HDD	13GB	204	141	141

We also compared the netsDB performance to TensorFlow, using the Keras implementation of the FFNN model. As illustrated in Tab. 8, 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.

Table 8: 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*

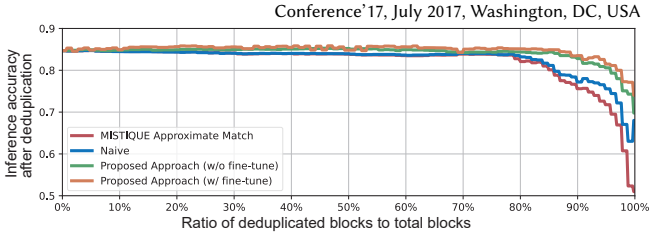

Figure 6: Comparison results of deduplicating a text classification model using different indexing approaches (block size: 100x10000)

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 [59]. As illustrated in Fig. 6, 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.

Moreover, we also compared the compression ratio, the average latency for querying one tensor block from the index, and the accuracy 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. 9 and Tab. 10, the proposed approach based on L2 LSH still achieved the best compression ratio. The Mistique's approximate approach 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 9: 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

Table 10: Comparison of model accuracy drop.

	Model-1	Model-2	Model-3	Model-4	Model-5
Mistique Exact Dedup	0.00%	0.00%	0.00%	0.00%	0.00%
Mistique Approximate Dedup	0.00%	0.00%	3.64%	4.06%	0.71%
Enhanced Pairwise	0.00%	0.00%	3.57%	3.58%	2.92%
Proposed (w/o finetune)	0.00%	0.00%	3.58%	3.59%	0.71%

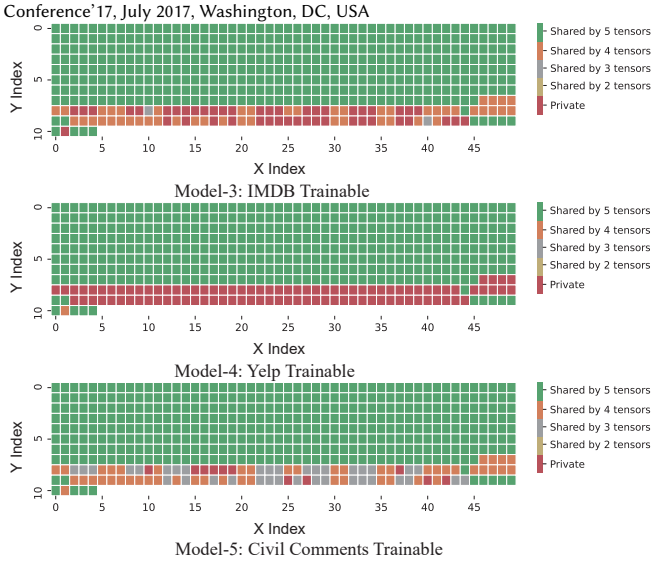

Figure 7: Block sharing in Text Classification

We also visualized the distribution of duplicate blocks across the models for the text classification workload, as illustrated in Fig. 7. The results showed that the blocks that are shared across models tend to be located in the same position of the tensor. This observation leads to the optimization of metadata as described in Sec. 3: metadata such as the index (i.e., position) of a shared tensor block in each tensor can be simplified.

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) A baseline algorithm, where we simply pack tensor blocks to pages in order, and then we eliminate the duplicate pages which contain the same set of tensor blocks.

We observed significant improvement in storage efficiency brought by our proposed two-stage algorithm compared to alternatives, as illustrated in Tab. 11. In addition, the computation efficiency of the two-stage algorithm is comparable to Greedy-1, as illustrated in Tab. 12. As mentioned, the extreme classification workload involves models that share the same fully connected layer, which means all tensor blocks in that layer are fully shared by all models. In such a special case, all algorithms achieve similar storage efficiency.

Table 11: Comparison of required number of pages using different page packing algorithms.

Scenario (block size, page size)	Baseline	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

Above testing results are based on the offline page packing. We also tested the online approach of page packing. We find that in the text classification workload, when using $100 \times 10,000$ block size and 64 megabytes page, each time we involve a new model, about 20% of pages need to be reorganized, while 80% of pages can be reused and thus do not need to be changed, as illustrated in Tab. 13.

Table 12: Comparison of page packing latency using different page packing algorithms. (Unit: seconds)

Scenario (block size, page size)	Baseline	Two-Stage	Greedy-1	Greedy-2
word2vec(100 × 10000, 64MB)	1.29	0.02	**0.01**	0.82
text classification (100 × 10000, 64MB)	0.68	**0.01**	**0.01**	0.52
text classification (300 × 300, 64MB)	13.65	**0.05**	**0.05**	11.50
text classification (300 × 300, 32MB)	44.72	**0.04**	**0.04**	42.72

Table 13: Page reuse and reorganization for online page packing.

Step	New model to pack	pages reused	pages discarded	pages created
1	Model-1	0	0	64
2	Model-2	52	11	15
3	Model-3	52	9	15
4	Model-4	50	13	23
5	Model-5	52	13	16

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-based page replacement policy without considering the page sharing. The detailed cache hit ratio comparison for the Word2Vec and text classification applications are illustrated in Tab. 8. Locality Set-M/L refers to the locality set page replacement policy [73, 74] 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.

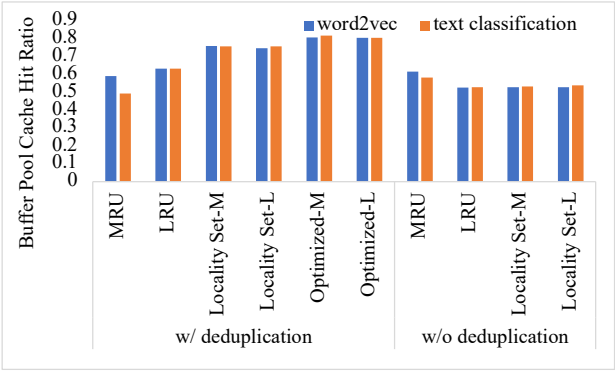

Figure 8: Comparison of different page replacement policies

7.6 Relationship to Model Compression

Besides deduplication, there exist a number of model compression techniques, such as pruning [27, 28] and quantization [33], 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 quantization individual models, which achieved 2× to 3× better storage efficiency. The reason is that pruning and quantization will not significantly change the similarity of tensor blocks across models.

We also observed similar results for an ensemble of VGG-16 models. We omit the details here, because the use cases of convolutional neural networks in RDBMS are unclear and the volume of model parameters is relatively small (up to hundreds of megabytes).

Table 14: 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

Serving deep learning models from RDBMS can greatly benefit from the RDBMS' physical data independence and manageability. This work proposed several synergistic storage optimization techniques covering indexing, page packing, and caching. We implemented the system 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, and (3) multiple extreme classification models specialized through transfer learning. 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.

REFERENCES

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

[2] [n.d.]. Implementation of Convolution as Matrix Multiplication. ([n. d.]). https://cs231n.github.io/convolutional-networks/

[3] [n.d.]. Optimizing TensorFlow Convolution Performance on Aarch64. ([n. d.]). https://www.linaro.org/blog/optimizing-tensorflow-convolution-performance-on-aarch64/

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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

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

[20] Edward G Coffman, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. Bin packing approximation algorithms: survey and classification. (2013), 455–531.

[21] 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)*.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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).

[28] 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).

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

[30] 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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, Yunseong Lee, and Byung-Gon Chun. 2020. Accelerating Multi-Model Inference by Merging DNNs of Different Weights. *arXiv preprint arXiv:2009.13062* (2020).

[36] 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).

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

[38] 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.

[39] Richard E Korf. 2002. A new algorithm for optimal bin packing. In *Aaai/Iaai*. 731–736.

[40] 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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *Proceedings of the 2021 International Conference on Management of Data*. 1222–1234.

[47] 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/

[48] 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.

[49] 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.

[50] 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.

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

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

[53] 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.

[54] 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).

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

[56] 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.

[57] Michael Sindelar, Ramesh K Sitaraman, and Prashant Shenoy. 2011. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 367–378.

[58] 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.

[59] 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.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[61] 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.

[62] 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).

[63] 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).

[64] 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.

[65] 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.

[66] 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).

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

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

[69] 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).

[70] 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.

[71] Jia Zou, Pratik Barhate, Amitabh Das, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chis Jermaine. 2021. Lachesis: Automated Generation of Persistent Partitionings for Big Data Applications. *Proceedings of the VLDB Endowment* (2021). "(To Appear)".

[72] 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.

[73] 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.

[74] 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.