NeuralStore: Efficient In-database Deep Learning Model Management System (Extended Version)

Paper No: 518

Abstract

With the prevalence of in-database AI-powered analytics, there is an increasing demand for database systems to efficiently manage the ever-expanding number and size of deep learning models. However, existing database systems typically store entire models as monolithic files or apply compression techniques that overlook the structural characteristics of deep learning models, resulting in sub-optimal model storage overhead. This paper presents NeuralStore, a novel in-database model management system that enables efficient storage and utilization of deep learning models. First, NeuralStore employs a tensor-based model storage engine to enable fine-grained model storage within databases. In particular, we enhance the hierarchical navigable small world (HNSW) graph to index tensors, and only store additional deltas for tensors within a predefined similarity threshold to ensure tensor-level deduplication. Second, we propose a delta quantization algorithm that effectively compresses delta tensors, thus achieving a superior compression ratio with controllable model accuracy loss. Finally, we devise a compression-aware model loading mechanism, which improves model utilization performance by enabling direct computation on compressed tensors. Experimental evaluations demonstrate that NeuralStore achieves superior compression ratios and competitive model loading throughput compared to state-of-the-art approaches.

CCS Concepts

• **Information systems → Database management system engines**; **Database design and models**.

Keywords

In-database Analytics, Deep Learning Model, Storage Engine

ACM Reference Format:

Paper No: 518. 2026. NeuralStore: Efficient In-database Deep Learning Model Management System (Extended Version). In *Proceedings of ACM International Conference on Management of Data (SIGMOD '26)*. ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern database management systems (DBMSs) are increasingly integrating artificial intelligence (AI) to support advanced data analytics [7, 15, 32, 33, 41, 43]. Such in-database AI-powered analytics enable users to issue complex data analytics tasks through specialized SQL interfaces [32, 46, 47]. DBMSs then automatically retrieve the relevant stored data and perform AI inference to provide deeper insights that traditional statistical operations (e.g., averages and sums) often fail to capture. As a result, the entire AI analytic workflow occurs within the database, which eliminates the need to move large amounts of data outside DBMSs, and thus facilitates efficient and secure analytics [15, 42].

Sectors such as finance [16, 20] and e-commerce [32, 33] are rapidly adopting in-database AI-powered analytics in their critical business workflows, and with the advancements of AI, deep learning (DL) models have become prevalent. Consider purchase recommendations in e-commerce as an illustrative example, where items are recommended based on personal user profiles, including habits, occupations, and lifestyles. As user profiles typically contain sensitive information, such as salary and browsing history, in-database analytics is particularly suitable for handling such recommendation tasks. To enable precise and personalized recommendations, e-commerce vendors commonly deploy specialized DL models tailored to different users, regions, or customer segments. These specialized models are often derived from fundamental pre-trained DL models [45], which may consist of dozens or even hundreds of layers, each potentially requiring gigabytes of storage [30, 38, 39]. Further, as user profiles continuously evolve over time, these specialized models must be regularly updated or fine-tuned, leading to a steadily increasing number of models. As a result, efficient in-database DL model management (i.e., storing and loading DL models directly within DBMSs) has become a foundational capability for in-database AI-powered analytics.

Existing in-database model management approaches generally treat each model as an isolated unit and store full-fledged models independently. For example, DBMSs such as PostgresML [7], Oracle [6], and Azure [4] serialize each model into a BLOB and store it directly in a dedicated model table [4, 6, 7]. Alternatively, systems such as ModelDB [39], RAVEN [33], and Vertica-ML [15] store file paths in the table, while placing the actual models as external files. Although straightforward, these methods suffer from substantial storage overhead, as storing hundreds of models can require terabytes of space due to redundant parameters and deep architectures. This overhead grows rapidly with the scale and complexity of DL model deployments. To mitigate storage costs, users can manually compress models before storing them into DBMSs using general data compression algorithms [2, 8, 26], floating-point compression schemes [26], or specialized model compression methods [38]. However, such optimization only partially addresses the storage issue, as they still handle each model independently, i.e., the overall storage cost remains proportional to the total number and size of models [38]. This persistent linear growth in storage overhead presents a critical bottleneck for scalable DL model management. Addressing this challenge requires new strategies that exploit structural similarities across models to reduce redundancy.

Storing a DL model involves two core components: 1) the model architecture, typically represented as a computational graph that defines the connectivity and operations of layers; and 2) a set of layers, where each layer comprises one or more high-dimensional floating-point tensors. As model fine-tuning is relatively common, many DL models share similar architectures and contain identical or highly similar tensors, particularly when fine-tuning is limited to a subset of layers [21, 30, 38]. This observation naturally motivates us to explore similarities and relationships across models, thus eliminating redundancy and improving overall storage efficiency.

Given that the model's learnable parameters (e.g., weights and biases) in tensors constitute the majority of a model's storage cost, we aim to identify redundant tensors and store only incremental differences between similar tensors. However, achieving this tensor-level deduplication requires addressing three key challenges: First, similarities between tensors are implicit. For instance, two models without explicit lineage may still contain similar tensors. Consequently, effectively identifying similar layers across a large collection of models is inherently challenging. Second, tensors typically consist of high-entropy floating-point parameters. Even if two tensors exhibit similar structures and parameters, directly calculating their parameter-wise differences can result in a delta tensor of identical dimensionality, yielding little to no storage savings. Thus, generating compact delta tensors that meaningfully reduce storage consumption is not straightforward. Third, since models are stored as fine-grained delta tensors, retrieving a model involves reconstructing the complete model based on these tensors. Due to the complexity and depth of DL models, this reconstruction process can be costly, and therefore, efficiently retrieving models is non-trivial.

In this paper, we present NeuralStore, an efficient in-database model management system designed to reduce DL model storage costs while streamlining model utilization. We first propose a tensor-based storage engine that departs from traditional per-model storage by identifying and storing shared tensor components across models, significantly improving space efficiency through structured deduplication. At its core is a high-performance tensor index built upon the Hierarchical Navigable Small World (HNSW) graph structure. Specifically, we categorize tensors into two types: *base tensors*, which store original parameters, and *delta tensors*, which maintain differences relative to a corresponding base tensor. Base tensors are stored as nodes in the HNSW-based tensor index, while delta tensors are placed separately in dedicated tensor pages. When saving a new model, we deconstruct it into individual tensors, and for each tensor, we search the tensor index to determine whether a similar base tensor already exists within a predefined similarity threshold. If such a base tensor is found, we compute and store the corresponding delta tensor; otherwise, unmatched tensors are stored as new base tensors. To integrate this design seamlessly into DBMSs, we build the tensor-based storage engine on top of modern database architecture, with enhancements specifically designed for efficient DL model management. In particular, we introduce a tailored index cache that efficiently buffers portions of the HNSW-based tensor index and extend the native page layout to support large tensors without disrupting the existing page-based storage mechanism.

We then introduce a delta quantization algorithm that compresses delta tensors to achieve reduced storage overhead. Unlike traditional quantization that operates on complete models, we quantize delta tensors, whose parameter ranges are typically much narrower than those of the original tensors. This mitigates the accuracy loss commonly associated with traditional model quantization. Moreover, our algorithm is adaptive, dynamically selecting the bit width for each delta tensor based on its parameter distribution, enabling fine-grained control over the trade-off between storage efficiency and model accuracy.

Lastly, we design a compression-aware model loading mechanism that enables direct computation on compressed tensors, eliminating the need to fully reconstruct models before use. Unlike traditional pipelines that first decompress models into memory, our approach integrates the reconstruction directly into the computation graph and pipelines tensor loading with computation. This tight integration reduces inference latency and memory overhead, improving model loading performance in in-database settings.

In summary, we make the following contributions:

- We present NeuralStore, a novel in-database DL model management system that enables efficient tensor-level storage and loading.
- We introduce a structured tensor-based storage engine that can be seamlessly integrated into modern DBMSs.
- We develop an adaptive delta quantization algorithm that minimizes storage by dynamically adjusting the bit width for each delta tensor.
- We design a compression-aware model loading mechanism that supports direct computation over compressed tensors, reducing the overall in-database AI-powered analytics latency.
- We implement NeuralStore as a pluggable PostgreSQL extension and evaluate its performance against state-of-the-art systems. Experiment results demonstrate substantial gains in end-to-end AI-powered analytics performance, storage efficiency, and model saving and loading throughput.
- We integrate NeuralStore into DuckDB, and the performance evaluation confirms its general extensibility.

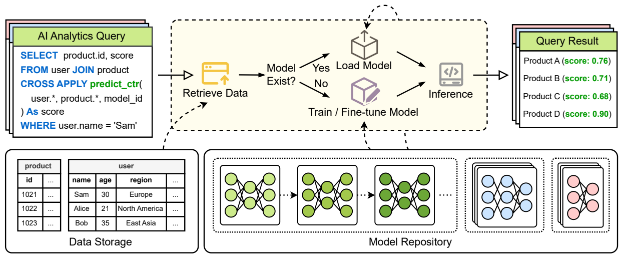
The remainder of the paper is structured as follows. Section 2 provides the relevant background and presents the problem statement. Section 3 overviews the system architecture of NeuralStore. Section 4 details the design of NeuralStore. Section 5 describes the system implementation, and Section 6 presents the experimental results. Section 7 reviews the related works, and finally, Section 8 concludes the paper.

2 Background

In this section, we provide the relevant background and formally define the problem that NeuralStore aims to address.

2.1 In-database AI-powered Analytics

In-database AI-powered analytics enables DBMSs to handle complex data analytics tasks via specialized SQL syntax [6, 32] or user-defined functions (UDFs) [7, 43]. For example, let us consider the click-through rate (CTR) prediction task, as shown in Figure 1. A data analyst submits a query to estimate CTR scores, i.e., the probability of a user clicking on the product. Upon receiving the query,



```
SELECT product.id, score
FROM user JOIN product
CROSS APPLY predict_ctr(
  user.*, product.*, model_id
) As score
WHERE user.name = 'Sam'
```

product	
id	...
1021	...
1022	...
1023	...

user			
name	age	region	...
Sam	30	Europe	...
Alice	21	North America	...
Bob	35	East Asia	...

Product A (score: 0.76)
Product B (score: 0.71)
Product C (score: 0.68)
Product D (score: 0.90)

Figure 1: In-database AI-powered Analytics Workflow – It consists of three main steps: (1) data retrieval, (2) model loading, and (3) inference.

the DBMS retrieves relevant data (e.g., product ID, user name, etc.) from the user and product tables, loads the DL model specified by the model ID, and performs model inference to predict CTR scores.

Since different analytics tasks often require different DL models, a model repository is typically maintained, where models are stored as files or binary large objects (BLOBs). Each model may have dozens or hundreds of layers and consume gigabytes of storage, and therefore, repeatedly loading these large models from disk for different tasks can introduce substantial latency. Moreover, with ongoing data updates, models need to be frequently retrained or fine-tuned to maintain accuracy. For instance, a CTR prediction model may generate multiple updated versions over time to reflect new user and product data, which significantly increases storage overhead. These factors limit the scalability of in-database AI-powered analytics and degrade the response time of analytics queries. To address these challenges, we design NeuralStore, an in-database model management system that reduces model storage overhead and improves model loading efficiency.

2.2 Storage Optimizations for DL Models

Existing storage optimization techniques for DL models can be broadly categorized into general-purpose data compression algorithms and model-specific compression techniques. General compression algorithms, such as ZSTD [8], can be directly applied to serialized DL model files, and are effective at removing exact duplicate patterns in the data. However, DL models rarely contain such duplicates [30, 38], especially across layers with diverse weights. Similarly, floating-point compression schemes such as ZFP [26] perform well on structured, spatially local data, but not on DL models whose weights are typically high-dimensional, continuous, broadly distributed, and lack spatial regularity. These characteristics limit the effectiveness of generic compressors.

Model-specific compression techniques address the aforementioned shortcomings. In particular, ELF [38] eliminates the exponent bits of floating-point values within the range $(-1, 1)$ by remapping them to the interval $[1, 2)$, to improve compressibility. Nonetheless, ELF operates on each model in isolation, missing opportunities to exploit shared structure across models.

Rather than optimizing each model independently, we aim to reduce storage costs by leveraging inter-model similarities and shared components. NeuralStore builds on this idea by enabling tensor-level deduplication, identifying and reusing redundant model components across a collection.

2.3 Hierarchical Navigable Small World

Approximate Nearest Neighbor search (ANN) is a technique for efficiently identifying data points in high-dimensional spaces that are approximately closest to a given query point [14, 40]. Existing ANN algorithms can be categorized into hashing-based [12, 22], tree-based [35], quantization-based [24, 48], and graph-based approaches [17, 29]. Among these, the Hierarchical Navigable Small World (HNSW) graph [29], a state-of-the-art graph-based method, is widely adopted due to its ability to achieve high recall with low query latency. HNSW organizes data points into a multi-layer graph structure. Each layer forms a navigable small-world graph, where nodes are connected to their approximate nearest neighbors. Higher layers provide coarse-grained shortcuts, while the lower layers enable fine-grained local search. Given a query, the search algorithm starts from a high-level node and performs greedy search layer by layer, descending through the hierarchy until it reaches the bottom layer, where it refines the search to identify the approximate nearest neighbor. Formally, let $G = (V, E)$ denote the HNSW graph, where V is the set of data points represented as vertices, and E consists of edges between points, weighted by their pairwise distances. Given a query point q and an entry point v_0, HNSW iteratively traverses the graph to find the neighbor v_{t+1} of the current vertex v_t that is closest to q. This process continues until no closer neighbor is found, at which point the current node is returned as the approximate nearest neighbor.

In NeuralStore, we leverage HNSW to index base tensors efficiently. This allows us to identify previously stored base tensors that are most similar to a given input tensor.

2.4 Post-training Model Quantization

Quantization is designed to reduce the computational and memory costs of DL models [18, 19, 25]. It maps a model's weights and activations to lower-precision formats (e.g., from Float32 to Int8). In general, quantization techniques can be classified into quantization-aware training (QAT) [13, 23] and post-training quantization (PTQ) [27, 31, 44]. QAT integrates quantization operations during model training, while PTQ is performed after model training. In PTQ, a Float32 tensor $\theta = \{x_1, x_2, ..., x_n\}$, where $x_i \in [x_{min}, x_{max}]$ is quantized to a tensor of b-bit integers $q_1, q_2, \ldots, q_n$ using a scale factor $s = \frac{x_{\max} - x_{\min}}{2^b - 1}$ and a zero-point offset z. Each value is quantized as $q_i = round\left(\frac{x_i - x_{min}}{s}\right) + z$. While PTQ is simple and efficient, it inevitably introduces precision loss, especially when the dynamic range of X is wide or the bit width b is small.

Unlike traditional use cases of PTQ, we leverage PTQ in the context of model management. We observe that delta tensors, i.e., the differences between similar tensors across models, typically exhibit smaller value ranges than the original tensors. As a result, applying PTQ to delta tensors can help mitigate the accuracy degradation typically associated with quantizing full tensors. To further reduce precision loss, NeuralStore dynamically adjusts the quantization bit width b for each delta tensor, based on its value distribution and a user-defined accuracy tolerance. This enables fine-grained compression while preserving model performance.

2.5 Problem Definition

Modern DBMSs increasingly support AI-powered analytics by incorporating DL models. In this context, in-database model management refers to the capability of DBMSs to store and utilize DL models efficiently. A well-designed system must achieve the following three objectives: (1) Storage consumption: Minimize the total storage required to maintain a collection of models, especially as the number and size of models grow. (2) Model accuracy: Preserve the predictive performance of models, particularly when storage-saving techniques (e.g., quantization or compression) are applied. (3) Query efficiency: Ensure high-throughput model loading and low-latency inference to support analytical workloads.

However, these objectives are often in tension with one another. For example, aggressive compression may reduce storage overhead, but at the cost of increased precision loss or additional decoding overhead during inference. Similarly, techniques that improve loading efficiency may require storing uncompressed or partially compressed models, thereby increasing storage consumption. Given these trade-offs, it is typically infeasible to optimize all three objectives simultaneously [43, 50]. In this work, we focus on minimizing storage consumption, which becomes increasingly critical as the number and size of models grow, while maintaining acceptable model accuracy and reasonable query efficiency. In particular, we ensure that the accuracy degradation caused by our approach remains within a user-defined tolerance, and the model is efficient for retrieval and inference. We now formally define the problem addressed by NeuralStore.

Problem definition. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ denote a set of n DL models, where each model M_i consists of a set of L_i layers, i.e., $M_i = \{\ell_{i,1}, \ell_{i,2}, \ldots, \ell_{i,L_i}\}$. Each layer $\ell_{i,j}$ contains a set of $K_{i,j}$ learnable tensors: $\ell_{i,j} = \left\{\theta_{i,j}^{(1)}, \theta_{i,j}^{(2)}, \ldots, \theta_{i,j}^{(K_{i,j})}\right\}, \quad \theta_{i,j}^{(k)} \in \mathbb{R}^{d_{i,j}^{(k)}}$, where $d_{i,j}^{(k)}$ refers to the dimensionality of the k-th learnable tensor in layer j of model M_i. Given a precision loss tolerance p (e.g., a relative or absolute error bound), the goal of NeuralStore is to jointly minimize the total storage cost and query latency of the DL model collection, while ensuring that the compression-induced accuracy loss remains within acceptable bounds. Formally, let $S(\mathcal{M})$ denote the total storage cost of the model collection $\mathcal{M}$, and $T(\mathcal{M})$ denote the total query latency (e.g., model loading and inference time). The optimization objective can be defined as:

$$\begin{aligned} \min \quad & \alpha \cdot S(\mathcal{M}) + \beta \cdot T(\mathcal{M}), \\ \text{subject to} \quad & \forall M_i \in \mathcal{M},\ \text{Error}(M_i) \leq p. \end{aligned} \tag{1}$$

Here, α and β are user-defined weights that balance the importance of storage efficiency and query performance. Error(M_i) represents the quantifiable accuracy degradation introduced by compressing model M_i, such as layer-wise tensor deviation or loss in downstream prediction accuracy.

3 System Overview

In this section, we describe the system architecture of NeuralStore. As shown in Figure 2, NeuralStore comprises three core components, namely the model compressor, the model loader, and the storage layer. Upon receiving users' *Save model* requests, the model compressor is invoked to reduce the model size using our delta quantization algorithm. The compressed tensor, updated ANN indexes, and model architecture are then serialized and stored in the storage layer. During model inference, users send *Load model* requests to the model loader, which fetches the compressed tensors from the storage layer and performs computation without full decompression. Next, we describe each component in detail.

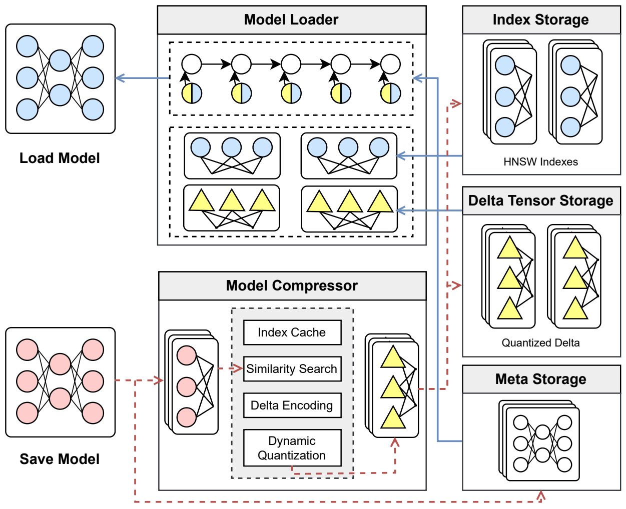


Figure 2: System Architecture of NeuralStore – The system supports two main workflows: (1) model saving (red), where models are compressed before being stored, and (2) model loading (blue), where models are retrieved by the model loader.

Storage Layer. The storage layer is responsible for managing all model-related data in NeuralStore, including HNSW, tensors, and model architectures. NeuralStore persists HNSW on disk, which are loaded into memory at runtime and used by the model compressor for efficient similarity search. To reduce the size of HNSW, we store the 8-bit quantized tensors in the vertices as the base tensors. NeuralStore stores model tensors as quantized deltas with respect to the corresponding base tensors. The quantization parameters, such as zero point and scale, are serialized and stored as the prefix of each quantized delta and tensor. Additionally, the serialized model architectures are stored in the meta storage. NeuralStore uses a relational table to organize model metadata, including model IDs and names, so that external users can easily manage and interact with models in the database.

Model Compressor. The model compressor is used to reduce the model size using our delta quantization algorithm (Section 4.2) according to the following steps. (1) The system first decouples model weights from the model architecture. This separation allows NeuralStore to manage weights at the granularity of individual tensors. (2) A similarity search is performed for each tensor to locate the most similar base tensor in the system. (3) The delta between the input tensor and the base tensor is then computed. (4) If the delta is sufficiently small to be quantized within the user-defined bit width, it is stored in the storage layer. Otherwise, a new HNSW vertex is created using the quantized value of the input tensor, and the process repeats from Step (2).

Model Loader. The model loader is designed to efficiently retrieve the required models using the compression-aware model loading

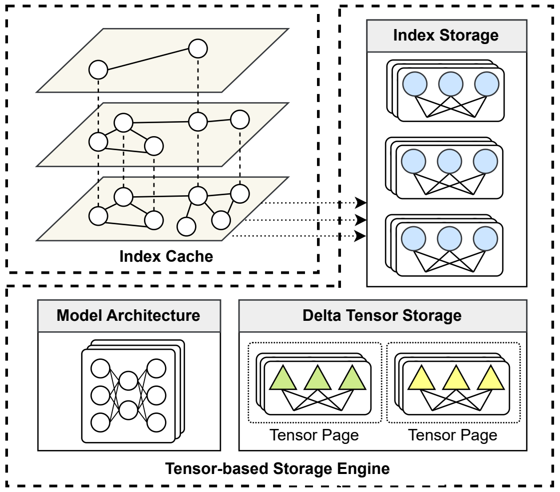


Figure 3: Tensor-based Storage Engine – The engine consists of three components: index storage for HNSW-based indexes, delta tensor storage, and metadata storage for model architectures.

mechanism. Specifically, NeuralStore loads the base tensors, delta tensors, and the computation graph into memory. To facilitate fast retrieval, tensors are loaded without full decompression, i.e., de-quantization and reconstruction. The shared base tensor is loaded only once, even when referenced by multiple layers or models. To reduce the memory usage, we modify the computation graph upon retrieval so that tensors are only de-quantized and reconstructed before they are invoked in the computation. NeuralStore then follows the modified computation graph to compute the results.

Running Example. Let us continue the CTR prediction example shown in Figure 1. After receiving the analytics task, the database retrieves relevant data and initiates a *Load Model* request to NeuralStore. In particular, NeuralStore employs the model loader to perform compression-aware loading, retrieving the associated tensors and computation graph for in-database CTR prediction. Further, when a newly trained or fine-tuned model needs to be stored, the database issues a *Save Model* request. In response, NeuralStore compresses the model using the adaptive delta quantization algorithm and persists the compressed tensors in the storage layer.

4 Design of NeuralStore

In this section, we detail the key techniques proposed for NeuralStore, including a tensor-based storage engine, a delta quantization algorithm, and a compression-aware model loading and inference mechanism.

4.1 Tensor-based Storage Engine

To exploit the structural similarities across DL models, NeuralStore organizes and compresses deep learning models at the granularity of individual tensors rather than entire models. This design enables similarity-based delta compression across models, allowing the system to avoid redundant storage by referencing previously stored tensors. The overall storage layout is illustrated in Figure 3. NeuralStore separates model storage into two main components: index storage, which stores shared base tensors used for reference, and delta tensor storage, which stores the differences between compressed tensors and their matched references. In addition, we serialize each model's architecture into a dedicated metadata storage, which extends the native tablespace that stores table structures commonly used in DBMSs.

Index Storage. Given that different models may contain tensors of varying shapes, NeuralStore maintains a collection of HNSW indexes, one per unique tensor structure (i.e., shape). Each HNSW index organizes similarly shaped tensors into a proximity graph, where each node stores a base tensor, and edges connect similar tensors to facilitate efficient ANN search. To reduce the index size, each base tensor is quantized to 8-bit using linear quantization prior to insertion. Although quantization introduces some loss, NeuralStore preserves full-precision representation recoverability by storing a corresponding delta tensor that captures the difference between the original tensor and its quantized representation. Our proposed delta quantization algorithm will be detailed in Section 4.2.

Delta Tensor Storage. The delta tensor storage is responsible for efficiently storing the compressed differences between base tensors and the tensors compressed relative to them. To support the typically large size of tensor data, we introduce a new page type in the database called a tensor page. Unlike standard heap pages, tensor pages are allowed to exceed the traditional page size limit and are managed separately to optimize read/write performance for large blocks. Within each tensor page, we store compressed deltas compactly. For each delta tensor, we store the following: 1) A 4-bit scale; 2) A 4-bit zero-point; 3) A quantized weight array. Each tensor is dynamically quantized based on its value range, and therefore, maintains its own scale and zero-point, which are used to de-quantize the delta tensor (Section 4.2). The quantized weights represent the difference between a base tensor and its corresponding variant, as determined by approximate similarity search. This design allows multiple models to share common base tensors while storing only the compressed deltas for fine-tuned variants. To further optimize model loading performance, delta tensors are organized in the order defined by the model architecture. This improves spatial locality and supports efficient reconstruction during model loading and inference.

Index Cache. To reduce the overhead of accessing HNSW in model compression and loading, we introduce an index cache that stores the deserialized HNSWs in memory. When a lookup of a base tensor is invoked, the system first checks the cache; if the corresponding HNSW index exists, the system bypasses disk I/O and the de-serialization process. This caching mechanism significantly reduces latency, particularly when models that share similar base tensors are loaded and saved frequently during iterative inference or fine-tuning. It maintains a bounded size and is managed using a least-recently used (LRU) eviction policy.

4.2 Delta Quantization Algorithm

Based on the properties discussed in Section 2.5, we propose a delta quantization algorithm to achieve efficient model compression while maintaining user-defined precision loss. Figure 4 depicts the workflow of model insertion. Given a collection of deep learning models $D = \{M_1, M_2, ..., M_n\}$, and a user-defined precision tolerance p, NeuralStore compresses the model as follows.

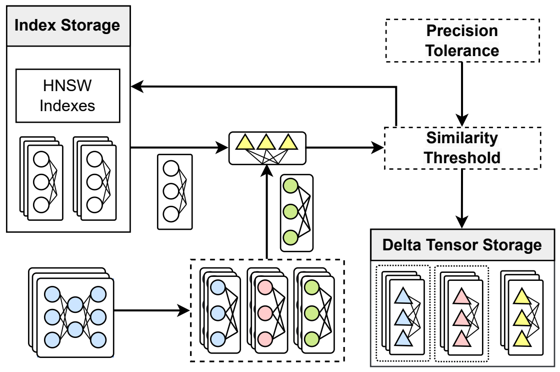


Figure 4: Delta Quantization Algorithm – NeuralStore compresses tensors in four steps: (1) decouple the weights from model architecture, (2) search for the closest base tensor with ANN, (3) perform delta-encoding, and (4) apply quantization to deltas.

Weight-Architecture Decoupling. Upon receiving the model saving request, we first decouple model weights (i.e., tensors) from model architectures to simplify and streamline the compression workflow. In NeuralStore, deep learning model architectures and tensors are managed independently. Given a set of deep learning models D, we extract their architectures into a set $S = \{s_1, s_2, ..., s_n\}$, and aggregate all tensors into a unified set $T = \{t_1, t_2, ..., t_m\}$, where m is the total number of tensors across all models. The architecture set S is stored in full, while compression is applied solely to the tensor set T. The benefit of such decoupling is that it enables independent optimization of the storage of tensors and architectures. In particular, we can flatten the tensors to one-dimensional arrays, so that they can match with more similar tensors for deduplication. For example, tensors with dimensions $(10, 10)$ and $(5, 20)$ are both flattened to a common shape of $(100, 1)$, increasing the opportunities of finding similar tensors.

Tensor Similarity Search. For each tensor $t \in T$, we search for a similar tensor already stored in NeuralStore using an approximate nearest neighbor index constructed for its shape. Specifically, we query the ANN index, A, to find a previously stored base tensor t_{base} that minimizes the similarity metric (e.g., Euclidean distance) with t. In our design, we use the HNSW index for its efficiency and strong performance in searching high-dimensional tensors, which are prevalent in deep learning models. As described in Section 4.1, the base tensor t_{base} stored in the HNSW vertex consists of quantized 8-bit integers. To enable the comparison with the input tensor, a 32-bit float, we de-quantize the base tensor, with the zero point and scale factor stored in the vertex, to $t_{full-base}$.

Delta Encoding. We calculate the delta encoding of the tensor, δ, and its bit width after quantization, denoted as $nbit$, as follows:

$$\delta = t - t_{full-base},$$
$$nbit = \left\lceil \log_2 \left(\frac{\delta_{max} - \delta_{min}}{2p} \right) \right\rceil. \tag{2}$$

where δ_{min} and δ_{max} are the minimum and maximum values in δ, respectively. We can observe that the range of δ and user-defined precision tolerance p collaboratively determine the final bit width to store the tensor. With a wide range of δ and a small precision tolerance p, the system will result in a large bit width and increase the storage consumption. Therefore, we introduce a threshold τ for the range of δ. If $\delta_{max} - \delta_{min}$ is less than or equal to τ, NeuralStore will proceed to quantize and store δ in the storage layer. Otherwise, NeuralStore creates a new vertex in HNSW and recalculates the delta based on the new vertex. This is to reduce the storage for the current tensor as well as potentially facilitate more effective compression for the future. The procedure is detailed as follows: (1) The original tensor t is quantized to obtain $t_{quantized}$, comprising 8-bit integers. (2) A new vertex storing $t_{quantized}$ is created and inserted into HNSW. (3) The system applies de-quantization to $t_{quantized}$ to get t'. The delta is calculated using $\delta = t - t'$. We have evaluated the impact of varying thresholds τ in Section 6.4.1. For a tensor with normalization, the range of the new delta falls within $\frac{1-(-1)}{2^8} \approx 0.0078$. According to our evaluation, it is recommended to set a threshold between 0.1 and 0.2 to achieve the best performance.

N-Bit Quantization. For tensors selected for compression, we quantize their delta values according to $nbit$ (calculated from Equation 2) to reduce storage cost while maintaining the precision loss within a user-defined precision tolerance p. To achieve this, we apply linear asymmetric quantization with a fixed bit width of $2p$, setting the scale accordingly:

$$quantized\_delta_i = \left\lfloor \frac{\delta_i}{scale} \right\rfloor + zero\_point, \tag{3}$$

where $scale = 2p$, and $zero\_point = \left\lfloor -\frac{\delta_{min}}{scale} \right\rfloor$. This indicates that the distance between two consecutive quantized numbers is $2p$, and therefore any points in between are within the distance of p to their closest quantized number. Moreover, since each tensor is individually quantized according to its value range, the number of bits required to store the tensor is minimized.

Algorithm 1 illustrates the complete model compression procedure. Given a set of deep learning models $D = \{M_1, M_2, \ldots, M_n\}$ and a user-defined precision tolerance p, NeuralStore first decouples each model into its architecture and tensors (Line 1). The set of architectures S is stored in full, while tensors T are compressed. For each tensor $t \in T$, the system performs an ANN search using an HNSW index to find the most similar base tensor t_{base} (Lines 3-4). If a sufficiently similar match is found, the system computes the delta $\delta = t - t_{\text{base}}$ (Line 5). Otherwise, t is quantized and inserted into the index, and the delta is recomputed against its quantized version (Lines 6-9). The delta tensor δ is then quantized using linear asymmetric quantization. The number of bits ($nbit$) is dynamically determined to ensure the quantization error remains within the tolerance p (Line 10). Each element of δ is quantized using the derived scale and zero-point (Lines 11-14). The quantized delta, along with the reference tensor and bit width metadata, is stored for future reconstruction (Line 15). Finally, the set of model architectures S is saved to complete the compression process (Line 16).

The delta quantization algorithm has three key novelties compared to existing methods. First, it exploits inter-model similarities. For each new model inserted, we search globally for the tensors closest to the input tensor, resulting in deduplication among models. Moreover, the integration of ANN enables flexible and incremental compression. As the system ingests more models, the growing number of base tensors increases the likelihood of finding close matches for newly added tensors, thereby improving compression efficiency

Algorithm 1: Delta Quantization in NeuralStore

```
Input: Model set $D = \{M_1, M_2, \ldots, M_n\}$, precision tolerance $p$
Output: Compressed tensors and model architectures stored in
        NeuralStore
1  $S, T \leftarrow$ DecoupleTensors($D$)   // Extract architectures $S$ and tensors $T$
2  foreach $t \in T$ do
3      $A \leftarrow$ GetIndexerFromPool($dim(t)$)
4      $t_{\text{base}} \leftarrow A$.Search($t$)
5      $\delta \leftarrow$ DeltaEncode($t, t_{\text{base}}$)
6      if ShouldCompress($\delta$) = false then
7          $t_q \leftarrow$ QuantizeForIndex($t$)
8          $A$.Insert($t_q$)
9          $\delta \leftarrow$ DeltaEncode($t, t_q$)
10     $nbit \leftarrow \left\lceil \log_2 \left( \frac{\max(\delta) - \min(\delta)}{2p} \right) \right\rceil$
11     scale $\leftarrow 2p$
12     zero_point $\leftarrow \left\lfloor -\frac{\min(\delta)}{\text{scale}} \right\rfloor$
13     foreach $\delta_i \in \delta$ do
14         qd$_i \leftarrow \left\lfloor \frac{\delta_i}{\text{scale}} \right\rfloor$ + zero_point
15     StoreQuantizedDelta(qd, $t_{\text{base}}$, $nbit$)
16 StoreArchitectures($S$)
```

over time. This approach eliminates the need to re-compress existing models and is particularly well-suited for dynamic model management scenarios where models are frequently added and fine-tuned. Second, it applies quantization to delta encoding. Compared with quantization over the original weights, this method substantially reduces the required bit width to represent model weights and lowers the precision loss by processing on a smaller scale. Lastly, each tensor and delta is dynamically quantized based on its value range, rather than applying a fixed global quantization parameter. This adaptive approach maximizes the compression ratio while adhering to the user-defined precision loss constraints.

Discussion. The effectiveness of the delta quantization algorithm is decided by the precision tolerance p, which defines the upper bound of the quantization bin width. A larger precision tolerance results in quantization with wider rounding intervals, thus yielding a higher compression ratio, but potentially degrading model performance. To mitigate such risk, NeuralStore uses a precision tolerance of 5.96×10^{-8} (2^{-24}) by default, which is smaller than the machine epsilon for single-precision floating-point numbers. As shown in Section 6.4.5, over 90% of the tested models exhibit no performance change under this tolerance, demonstrating that the default precision tolerance is sufficiently strict to limit the impact on model performance. NeuralStore also allows users to configure the precision tolerance on a per-model basis. We provide a utility tool in our code repository [10] to guide users in selecting an appropriate tolerance for a specific model. First, given a model and a test dataset, it compresses the model using multiple candidate tolerances. Next, it evaluates the performance of each compressed variant on the test data and reports the results to the user. Based on the analysis results, users can choose a preferred tolerance to store the model accordingly, effectively balancing storage consumption and model performance degradation.

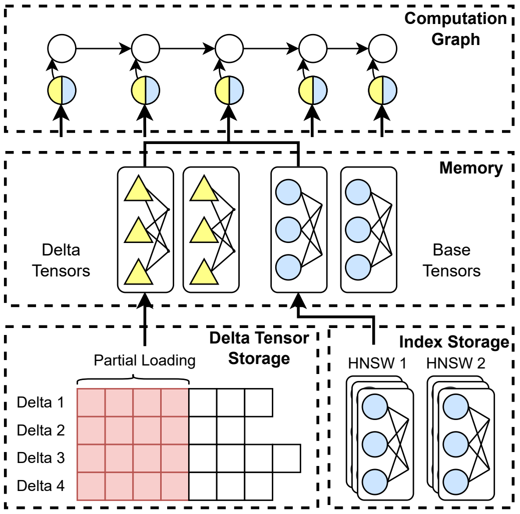


Figure 5: Compression-aware Model Inference – NeuralStore adopts flexible tensor loading with partial delta tensor bits and on-demand decompression to streamline the model loading process.

4.3 Compression-aware Model Loading

In conventional model management systems, compressed models are often required to be decompressed or reconstructed before they are served for inferences. This results in significant overhead in model loading and extensive memory consumption, accommodating the full model inside memory. As shown in Figure 5, we present the compression-aware model inference mechanism, which streamlines the model loading and inference process, as well as reduces the memory consumption.

4.3.1 *Model Loading.*

When a *Load Model* request is received, NeuralStore first looks up the reference of the first tensor page in the model table with the model ID. Since the tensor pages of a model are organized consecutively, it then scans the delta pages for model architecture, delta tensors, and references (HNSW ID and vertex ID) to base tensors. Lastly, NeuralStore traverses the HNSWs to fetch the index pages containing base tensors. NeuralStore only stores the quantized tensors in memory to reduce the memory consumption.

Flexible Model Loading. NeuralStore enables flexible model loading to optimize the trade-off among memory consumption, loading efficiency, and precision loss. In scenarios where the efficiency of model serving is critical, while higher model tolerance is accepted, NeuralStore allows the users to selectively fetch partial bits of delta tensors, or even only the base tensor. This will lead to faster model loading and lower memory consumption due to fewer bits loaded and disk I/O at the cost of higher model precision loss. Notably, the additional precision loss brought by the flexible model loading only has a limited impact on the resulting model performance due to our unique compression algorithm, as shown in Section 6. Since each delta is calculated with respect to the closest base tensor, and quantization is applied dynamically on each delta, ignoring the least significant bits of the quantized delta leads to an average difference of 10^{-4} compared with fetching the quantized delta in full bits.

Algorithm 2: Compression-Aware Model Inference

```
   Input: Compressed model $\hat{M}$, inference bit width $b$
   Output: Augmented computation graph $G$ for runtime execution
1  $G \leftarrow$ LoadModelGraph($\hat{M}$)
2  $T \leftarrow$ GetCompressedTensors($\hat{M}$)
3  foreach tensor $t \in T$ do
4      $t_{\text{base}} \leftarrow$ RetrieveQuantizedBase($t$)
5      $t_{\text{delta}}, nbit \leftarrow$ RetrieveQuantizedDelta($t$)
6      if $nbit > b$ then
7          $t_{\text{delta}} \leftarrow$ ExtractMSB($t_{\text{delta}}, b$)
8          $\text{scale}_{\text{delta}} \leftarrow \text{scale}_{\text{delta}} \times 2^{nbit-b}$
9      $N_{\text{deq\_base}} \leftarrow$ CreateDequantizeNode($t_{\text{base}}, \text{scale}_{\text{base}}, \text{zp}_{\text{base}}$)
10     $N_{\text{deq\_delta}} \leftarrow$
         CreateDequantizeNode($t_{\text{delta}}, \text{scale}_{\text{delta}}, \text{zp}_{\text{delta}}$)
11     $N_{\text{add}} \leftarrow$ CreateAddNode($N_{\text{deq\_base}}, N_{\text{deq\_delta}}$)
12     $G$.InsertNode($N_{\text{add}}$)
13     $G$.DirectOutput($N_{\text{add}}$, OriginalNode($t$))
14 return $G$
```

4.3.2 On-demand Decompression. NeuralStore adopts the on-demand decompression during model serving to ensure the minimum memory usage. To serve the model, NeuralStore first deserializes the model architecture to form a computation graph. When the computation reaches the step that involves a compressed tensor, it will de-quantize the delta tensor and the corresponding base tensor and reconstruct them to get the full-bit-width tensor for calculation. The full-bit-width tensor will be discarded after the computation is finished. This leads to consistent memory usage with additional computation cost for decompression. Such overhead can be mitigated by temporarily storing the de-quantized base tensors, which will be used again by the following layers. In particular, during model loading, we record the share count of each base tensor. For base tensors with the share count greater than 0 during the computation, we store the de-quantized base tensor and decrease the share count. When the share count reaches 0, the de-quantized base tensor will be deleted. In this way, we eliminate the duplicate de-quantization of the same base tensors.

Augmented Computation Graph. To enable on-demand decompression inference, NeuralStore augments the original model graph with additional computational nodes that handle dequantization and reconstruction at runtime. Specifically, a compressed tensor is reconstructed by combining two components: the quantized base tensor and its corresponding quantized delta. The base tensor is always stored and loaded in 8-bit quantized form, while the delta tensor is flexibly loaded based on the desired inference precision, as discussed in Section 4.3.1. Both the base and delta tensors are dequantized using their associated scale and zero-point values, which are retrieved alongside the quantized representations. These dequantization operations are expressed as `DequantizeLinear` nodes within the computation graph. The outputs of the two dequantization branches are then combined through an element-wise addition node to reconstruct the original tensor. This augmented graph eliminates the need for full offline decompression and enables efficient execution directly over the compressed representation.

We illustrate the compression-aware model inference process in Algorithm 2. Given a compressed model $\hat{M}$ and the targeted delta inference bit width b, NeuralStore first loads the original computation graph (Line 1). For each tensor, the system retrieves its quantized base and delta components, along with their associated quantization metadata (i.e., bit width, scale, and zero-point) (Lines 4-5). If the delta tensor was originally quantized to a higher bit width than the target delta bit width b, only the most significant b bits are extracted. To compensate for the bit truncation, the quantization scale is adjusted proportionally (Lines 6-8). The system then creates two `DequantizeLinear` nodes, one for the base tensor and one for the (truncated) delta tensor (Lines 9-10). These two dequantized outputs are fused through an `Add` node to reconstruct the tensor in float space (Line 11). NeuralStore inserts the dequantization and addition nodes directly into the model graph. The output of this reconstruction subgraph is then wired to the corresponding original node that consumed the tensor (Lines 11-13). This augmentation enables runtime reconstruction of compressed tensors and eliminates the need for full offline decompression.

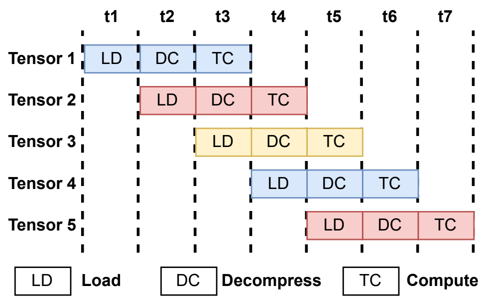


Figure 6: Pipelining – NeuralStore pipelines tensor loading, decompression, and computation during model loading.

4.3.3 Pipelining. We further improve the model serving process by leveraging pipelining. The entire process can be divided into three phases, namely, model loading, tensor decompression, and model computation. The model loading phase is I/O-intensive, as it involves retrieving delta tensors and HNSW indices that store the base tensors. Tensor decompression is primarily CPU-bound, though its computational overhead can be mitigated through the use of AVX instruction sets. Model computation is also CPU-intensive, dominated by matrix multiplication operations. To improve throughput, we pipeline these three phases. As illustrated in Figure 6, at the i-th stage of the pipeline, the system performs model loading for the i-th tensor, decompression for the $(i-1)$-th tensor, and model computation for the $(i-2)$-th tensor. This design enables concurrent execution of the three phases, effectively hiding latency and improving resource utilization. The benefits of pipelining are further amplified when the model computation phase is offloaded to a GPU due to less resource contention.

5 Implementation

We implement NeuralStore as a pluggable extension of PostgreSQL with 5000 LoC lines of code in C/C++. We have made our source code available [10].

NeuralStore is tightly integrated with PostgreSQL to support efficient model storage, loading, and compression-aware inference through SQL interfaces. We provide several PostgreSQL UDFs, such as `ns_save_model` and `ns_load_model`, to store and retrieve models. NeuralStore manages model storage at the granularity of tensors. Compressed tensors are organized into *tensor pages*, each of which contains the complete set of compressed tensors for a single deep learning model. Tensor pages are enforced to be read-only by NeuralStore, and are memory-mapped using `mmap` to reduce memory footprint and enable sharing across PostgreSQL sessions. At the start of each tensor page, a fixed-length header records the offsets and lengths of all delta tensors. Each delta tensor keeps metadata, including its shape, dimension, quantization parameters (i.e., scale and zero point), and single-element bit width, followed by a bit-packed payload.

To accelerate the retrieval of HNSW indexes, NeuralStore maintains a local cache for HNSW indexes. The cache is bounded by a user-defined buffer size, which is set to 32 GB by default at the startup stage of PostgreSQL. When memory is insufficient, the least recently used index is evicted from the cache to the disk based on LRU. We build our index on top of hnswlib, and extend a new `SpaceInterface` called `QuantizedL2Space` to support distance computation between quantized vectors with different scales and zero-points. To further accelerate this, we optimize the distance computation using AVX2 SIMD instructions.

NeuralStore supports both full decompression and compression-aware inference for ONNX framework models. For compression-aware inference, we modify ONNX computation graphs to incorporate on-demand decompression. Particularly, the system retrieves quantized delta and base tensors and inserts `DequantizeLinear` and `Add` nodes into the graph to perform runtime reconstruction, as described in Section 4.3.2. During compression, both delta computation and quantization are carried out in double precision to mitigate rounding errors introduced by low-precision representations.

To demonstrate the generalizability of the proposed method, we also implement NeuralStore as a loadable extension of DuckDB. Adapting NeuralStore to DuckDB involves three customizations: (1) rewriting the UDF registration logic, (2) replacing PostgreSQL's shared-memory-based index cache with an in-process index cache, and (3) adapting memory allocation from PostgreSQL to DuckDB's C++ runtime. This DuckDB-based implementation is functionally comparable to the PostgreSQL-based implementation, enabled by two key design factors. First, NeuralStore relies on components that are commonly available in modern DBMSs, such as page-based storage engines, UDF interfaces, and buffer managers. Second, its modular design decouples database-specific APIs, such as shared memory management and data access methods, from the core logic of model compression and loading.

6 Performance Evaluation

In this section, we evaluate NeuralStore by comparing it with state-of-the-art model management systems and compression algorithms. We conduct system-level and micro benchmarks to measure the query throughput, storage consumption, and model accuracy.

6.1 Experimental Setup

We conduct our experiments on two servers, each equipped with an Intel(R) Xeon(R) W-1290P CPU (10 cores × 2 hardware threads), 128GB of DRAM, a 894GB SAMSUNG_MZ7L3960 SSD, and an NVIDIA RTX 3090 GPU. We use one server to simulate clients that issue save model and load model requests, while the other server functions as the database server.

Table 1: Experimental AI-powered Analytics Workloads

ID	Workload	DataSet	Model
(a)	Sequence Classification	IMDB	DistilBERT
(b)	Image Classification	Beans	Vision Transformer
(c)	Tabular Classification	Avazu	MLP

Table 2: Summary of models used in the workloads

Type	Model	Count	Size (GB)
CV	MobileNetV2	30	0.25
	ResNet-50	110	9.63
	ViT-B/16	110	35.43
	ViT-L/32	5	5.71
	Swin-T	70	7.35
	Swin-B	60	19.74
NLP	BERT-base	50	20.41
	DistilBERT	50	12.37
	RoBERTa	50	23.23
	BERT-large	50	62.39
	T5-small	50	11.28
	T5-base	50	41.54
	BART-base	50	25.98
	BART-large	50	75.71
Multimodal	BLIP-base	15	10.97
Total		**800**	**361.99**

6.1.1 Workloads. To evaluate model management performance under realistic scenarios, we collect 800 DL models totaling 361GB, covering a diverse range of model architectures and sizes. Due to the space constraints, we provide a summary of these models in the extended version [9]. Before running the experiments, we perform a warmup phase to download all models from Huggingface [5] and store them locally in advance. We feed the systems with read and write workloads, containing randomly selected models, to simulate model saving and loading operations. We first save the models into the evaluated system to measure write throughput and subsequently load them to measure read throughput.

As summarized in Table 2, we set up three representative in-database AI-powered analytics tasks: (a) sequence classification [11]: 20 DistilBERT models fine-tuned on the IMDB dataset [28], each performing 100 text inferences to classify movie reviews, (b) image classification [43]: 20 Vision Transformer (ViT) models fine-tuned on the Beans dataset [1], each processing 100 images to identify bean leaf diseases, and (c) tabular classification [49]: 12 multi-layer perceptron (MLP) models trained on the Avazu dataset [3], each classifying 500 user records to predict CTR scores.

6.1.2 Baselines. We compare it against two representative baseline systems: a database-based model management system, PostgresML, and a file-based model management system, ELF*. To facilitate fair comparisons, all systems are integrated with PostgreSQL to store the metadata of models. We employ an open-sourced implementation on PostgreSQL [49] that provides standard in-database AI-powered analytics interfaces, and integrate it with the evaluated systems to perform end-to-end experiments, i.e., conducting model saving, model loading, and model inference within the database.

PostgresML [7]. PostgresML is a PostgreSQL extension designed for in-database machine learning. It stores each model as a serialized BLOB in a model table and leverages PostgreSQL's built-in TOAST mechanism for compression, using a LZ-family compression algorithm known as PGLZ.

ELF* [38]. To the best of our knowledge, there is no open-source file-based model management system. We therefore implement ELF*, a file-based baseline that integrates ELF [38], a state-of-the-art model compression algorithm. When saving new models, ELF* compresses each model using ELF and stores it as a separate file, and then records the file path in a PostgreSQL model table. When loading a model, ELF* fetches the model metadata (including the file path) from the database, loads the model file from disk, and decompresses it for use.

We also compare NeuralStore with widely used general-purpose compression algorithms, ZSTD and ZFP, and specialized model compression methods, ELF, in terms of storage consumption and the resulting model accuracy.

ZSTD [8]. ZSTD is a lossless compression algorithm developed by Facebook. It is commonly used for general-purpose data compression. We use its official release (v1.5.5) in our experiments.

ZFP [26]. ZFP is a lossy compression algorithm designed for floating-point arrays. It allows users to adjust error bounds according to the accuracy requirements of the target data. We use its official release (v0.5.5) to conduct our experiments.

ELF [38]. ELF is a state-of-the-art model compression framework that eliminates the exponent fields of floating point numbers by projecting the model weights from (-1,1) to [1, 2). We use its official open-source code and extend it to support the ONNX model format to align with our workload.

6.1.3 Default configuration. We configure PostgreSQL with a 32GB shared buffer. Other PostgreSQL parameters remain at their default values unless specified. We choose $\tau = 0.16$ as the default similarity threshold in NeuralStore, and enable the flexible model loading by default. We set the precision tolerance of ZFP and NeuralStore to 5.96×10^{-8}, consistent with that used in ELF, ensuring a uniform upper bound on the precision loss. A detailed analysis of how these default parameter settings are determined is provided in Section 6.4.

6.2 System Performance Evaluation

Here we evaluate the system performance of NeuralStore by comparing it with PostgresML and ELF*.

6.2.1 End-to-end Performance Evaluation. We first evaluate the end-to-end latency for in-database AI-powered analytics. In each task, we save multiple models as described in Section 6.1, then load each model and perform inference. We accumulate the latency of each stage and show the results in Figure 7. NeuralStore reduces total query latency by up to 32%, 26%, and 47% compared to PostgresML, and by up to 20%, 22%, and 14% compared to ELF*, across sequence, image, and tabular classification tasks. For model saving, NeuralStore takes 46s, 60s, and 5s for the three tasks, outperforming both PostgresML (71s, 82s, 14s) and ELF* (52s, 70s, 6s). This improvement is attributed to our tensor-based storage engine, which batches tensor writes into pages and defers HNSW index updates until eviction, thereby reducing I/O overhead. For model loading, NeuralStore achieves up to 61% and 50% speedup over ELF* and PostgresML, respectively. This is due to the fact that our tensor-based storage engine avoids redundant base-tensor fetching and reduces disk I/O. Moreover, our compression-aware model loading mechanism eliminates decompression during loading, further improving the model loading throughput. Inference latency remains comparable across all systems since they share the same ONNX runtime, with only negligible increases for NeuralStore due to the on-demand decompression during inference.

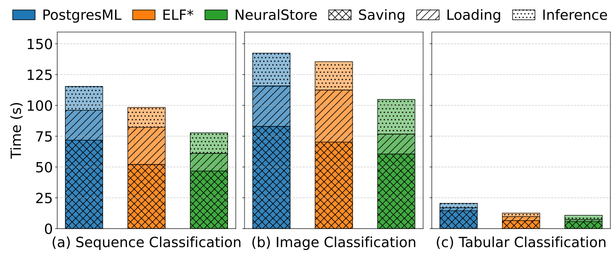


Figure 7: End-to-end Time Breakdown for In-database AI-powered Analytics

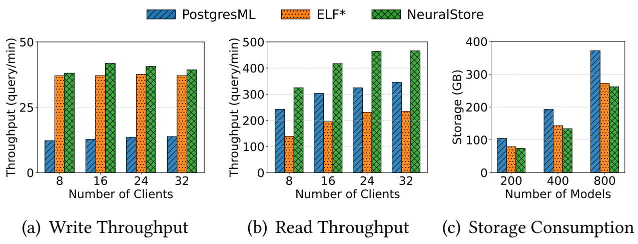


Figure 8: Overall Performance of NeuralStore.

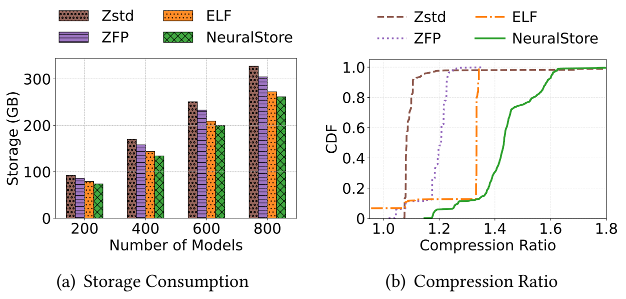


Figure 9: Evaluation on Compression Algorithms.

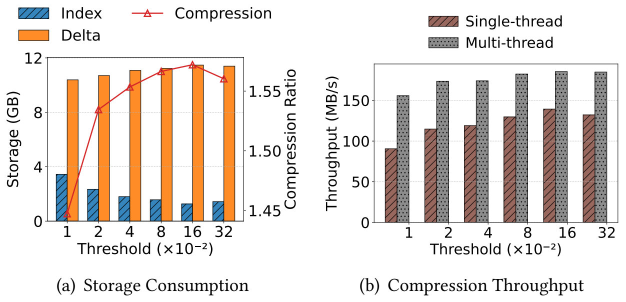


Figure 10: Performance Impact of Similarity Threshold.

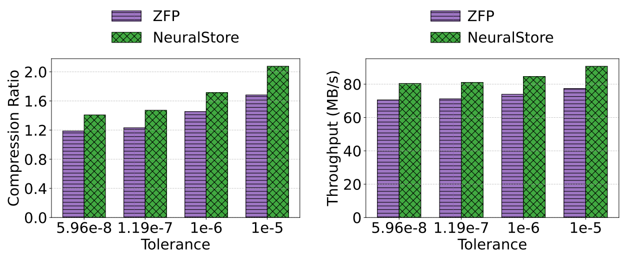
6.2.2 Throughput. We then evaluate the write and read throughput with varying numbers of clients ranging from 8 to 32. For write operations, the clients concurrently save models randomly chosen from the model pool described in Section 6.1.1. The results are shown in Figure 8(a). We can observe that the throughput of all systems increases as more clients are included due to the increased concurrency. NeuralStore outperforms the baselines, achieving a peak throughput of 42 queries per minute compared with 37 for ELF* and 17 for PostgresML. PostgresML performs the worst by storing the model with the TOAST mechanism in PostgreSQL. The models are divided into small chunks, which are fetched separately and reconstructed. This leads to high disk I/O overhead. ELF* achieves throughput comparable to NeuralStore across all numbers of clients, but remains behind due to higher I/O cost, especially when multiple models are stored concurrently. For read operations, each client continuously sends queries, with each query loading a random model. Figure 8(b) shows the read throughput. NeuralStore outperforms ELF* and PostgresML by up to 2.5× and 1.4×, respectively. The improvement in the performance stems from three key designs. First, NeuralStore maintains an in-memory index cache to avoid repeated fetching of base tensors, thus reducing the disk I/O. Second, NeuralStore adopts an on-demand decompression strategy, where quantized deltas and base tensors are de-quantized at inference time, thereby eliminating the need for full model decompression prior to execution. Lastly, the flexible model loading enables NeuralStore to load only the most significant 8 bits of the quantized deltas for inference, further reducing the disk I/O and memory bandwidth.

6.2.3 Storage. We report the resulting storage usage in Figure 8(c). As observed, NeuralStore has the least storage consumption. In particular, it consumes 93% and 70% of the space required by ELF* and PostgresML, respectively. Overall, NeuralStore achieves a compression ratio of 1.38×, compared to 1.32× for ELF* and 0.97× for PostgresML. These improvements are due to NeuralStore's delta quantization compression, which identifies shared base tensors across models and dynamically quantizes the base and delta tensors to reduce storage cost.

6.2.4 In-depth Bottleneck Analysis. To better understand system bottlenecks, we report CPU and I/O costs for saving and loading a representative model (google/vit-base-patch16-224) in Table 3. For model saving, NeuralStore achieves the shortest wall time and lowest I/O block usage. This aligns with the results in Figure 8(a). In addition, NeuralStore exhibits the highest CPU utilization compared to PostgreSQL and ELF*, which is expected because its higher compression ratio incurs greater computational cost. For model loading, NeuralStore achieves up to 50% and 55% lower wall time than PostgresML and ELF*, respectively. The results are also consistent with its higher read throughput shown in Figure 8(b). We also observe that NeuralStore achieves the lowest system time, CPU utilization, and I/O block reads. This is attributed to the on-demand decompression and flexible loading strategies, which reduce the cost of fully decompressing models and I/O overhead. In addition, we measure the memory usage of the evaluated model. The results show that NeuralStore consumes 165MB of memory during model loading, compared to 330MB for both PostgresML and ELF*. This further demonstrates the effectiveness of the proposed flexible loading mechanism in reducing memory consumption.

Table 3: CPU and I/O Statistics for Model Saving / Loading

Operation	System	Wall Time (s)	User Time (s)	System Time (s)	CPU Utilization	I/O Blocks
Model Saving	PostgresML	4.828	2.491	1.276	0.780	1154816
	ELF*	4.794	2.783	1.633	0.921	506776
	NeuralStore	3.283	2.087	0.971	0.932	348504
Model Loading	PostgresML	1.919	0.503	1.308	0.944	702680
	ELF*	1.982	1.395	0.583	0.998	507192
	NeuralStore	0.895	0.470	0.227	0.779	348568



(a) Compression Ratio (b) Compression Throughput

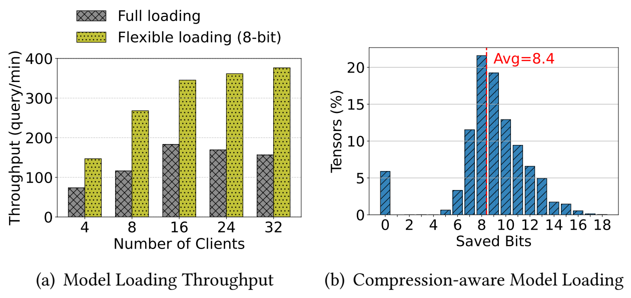
Figure 11: Performance Impact of Precision Tolerance.

6.3 Compression Performance Evaluation

We now assess the compression performance of NeuralStore. We conduct the experiments by progressively increasing the number of stored models from 200 to 800, and measure the storage consumption. Moreover, we calculate the compression ratio according to the original size, totaling 361GB.

6.3.1 Storage and Compression Ratio. We plot the storage sizes of the compressed models obtained using different compression algorithms in Figure 9. NeuralStore consistently achieves the lowest compressed size across all model scales. At 800 models, it reduces the total storage to 261GB, corresponding to a compression ratio of 1.38×. While the compression ratios for ELF, ZFP, and ZSTD are 1.32×, 1.18×, and 1.10×, respectively. This trend persists across different scales, demonstrating the scalability and effectiveness of delta quantization compression.

6.3.2 Per-model Compression Ratio Distribution. We study the per-model compression effectiveness of NeuralStore. To account for shared base tensors, we evenly distribute the storage cost of each base tensor in the index across all tensors that reference it. Figure 9(b) shows the cumulative distribution function (CDF) of per-model compression ratios. NeuralStore outperforms all baselines across the distribution. Over 60% of models achieve a compression ratio greater than 1.4×, and nearly 90% exceed 1.3×. In contrast, no model compressed with ELF reaches 1.4×, and fewer than 3% of models do so with ZFP or ZSTD. We also observe that the three baseline methods exhibit steep CDF curves concentrated between 1.2× and 1.3×, which indicates limited variability in their compression effectiveness. For example, ELF compresses tensors by deduplicating the 8-bit exponent field of floating-point values, which caps its ideal compression ratio around 1.33×. In contrast, NeuralStore exploits tensor-level similarities across models, enabling adaptive compression that achieves higher ratios when redundancy is present.



(a) Model Loading Throughput (b) Compression-aware Model Loading

Figure 12: Performance Impact of Flexible Model Loading.

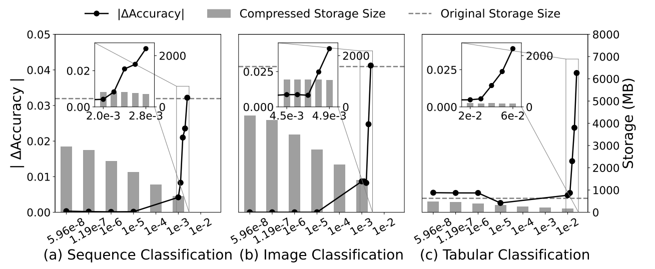


Figure 13: Model Accuracy and Storage Change under Different Precision Tolerance

6.4 Micro-Benchmarks

To gain deeper insights into the performance trade-offs, we conduct micro-benchmarks to evaluate the impact of key system parameters, namely, the similarity threshold τ, the user-defined precision tolerance p, as well as the design choice of flexible model loading.

6.4.1 Performance Impact of Similarity Threshold. A key parameter that influences the storage cost in NeuralStore is the similarity threshold (τ), which decides whether a tensor can be delta-compressed depending on its distance from base tensors. A higher threshold enables more aggressive delta compression by accepting looser matches, thereby reducing the number of base tensors added to the HNSW indexes. However, this also results in degraded effectiveness of delta-encoding. We evaluate the impact of τ using a subset of the dataset, consisting of 50 DL models fine-tuned from `google/bert-base`. We vary the similarity threshold and measure the resulting storage sizes of delta tensors and HNSW indexes.

Figure 10(a) shows how varying τ affects storage consumption. As τ increases, more tensors are qualified for delta encoding. Consequently, it reduces the opportunity of creating new vertex nodes in HNSW indexes, resulting in smaller index sizes. However, the decrease in the storage of HNSW indexes slows down when τ increases beyond 0.16. This is because HNSWs still need to maintain a minimum number of vertices for excessively distant tensors. Similarly, the storage of delta tensors increases when τ is small, while the marginal increase diminishes as τ becomes higher. It is because a higher similarity threshold results in a delta computed against sub-optimal base tensors, and therefore increases the number of bits required to represent the tensor. When τ exceeds a certain range, in this case is 0.16, the allowed distance is greater than the nearest base tensors. As a result, tensors are always able to find another base tensor to create a smaller delta. Therefore, the increase in the delta storage diminishes. The overall compression ratio (plotted as a red line) reflects the trade-off between index storage and delta storage. In the beginning, the index storage drops faster, leading to an increased compression ratio. It peaks at $\tau = 0.16$. After that, the increase in delta tensors storage overwhelms the space saved by indexes, leading to compression ratio drops.

We also evaluate the impact of τ on compression throughput under single-threaded and multi-threaded settings. For the multi-threading setup, we perform compression on the same 50 BERT models using two threads. Each thread fetches the tensor to be compressed from a shared queue and performs the similarity search, delta encoding, and quantization independently. The results are shown in Figure 10(b). As the similarity threshold τ becomes higher, the compression throughput increases from 90.5MB/s to 139.3MB/s for single-thread execution, and 155.8MB/s to 197.9MB/s for multi-thread execution. This trend aligns with earlier observations, as more tensors are delta-compressed instead of inserted into the index, the system avoids costly HNSW index maintenance, leading to faster overall compression. Notably, when $\tau = 0.16$, NeuralStore achieves the highest throughput and compression ratio.

6.4.2 Performance Impact of Precision Tolerance. We now evaluate the effectiveness of the delta quantization algorithm under varying precision tolerance p. The precision tolerance is given by users as a parameter when storing each model. It defines the upper bound of the quantization bin width, ensuring the resulting model accuracy is not significantly compromised. Varying the precision tolerance enables users to balance the trade-off between compression performance and model accuracy. In this experiment, we vary the precision tolerance from 5.96×10^{-8} (single precision machine epsilon) to 10^{-5}, and compare the compression ratio and throughput of NeuralStore and ZFP on the full 800-model set we collected.

The compression ratios are shown in Figure 11(a). As the precision tolerance increases, the compression ratio also improves because a wider tolerance (larger bin width) reduces the number of bits required during quantization. NeuralStore consistently outperforms ZFP across all tolerance levels. For example, at a tolerance of 10^{-5}, the compression ratio of NeuralStore is 2.07× and 1.68× for ZFP, exhibiting a 1.2× improvement. This is attributed to NeuralStore's ability to exploit inter-model tensor similarity, which becomes more effective as the precision tolerance increases.

We further measure the compression throughput with respect to the range of precision tolerance. As illustrated in Figure 11(b), the throughput of NeuralStore increases from 80.4MB/s to 90.7MB/s as the tolerance varies from 5.96×10^{-8} to 10^{-5}. This is because as precision tolerance increases, the number of bits to represent the delta becomes fewer. Consequently, more tensors will fall within the similarity threshold, resulting in fewer tensors to be inserted into the HNSW indexes. By reducing the costly index insertions and graph maintenance operations, the overhead of delta quantization compression is significantly mitigated. As a result, NeuralStore achieves faster compression rates at higher tolerance levels, without sacrificing its compression advantage.

6.4.3 Performance Impact of Flexible Model Loading. We evaluate the performance impact of flexible model loading, which provides users with options to load quantized-delta in full bit width and partial bit width. To quantify the trade-off between model accuracy and efficiency in both model loading and memory usage, we conduct

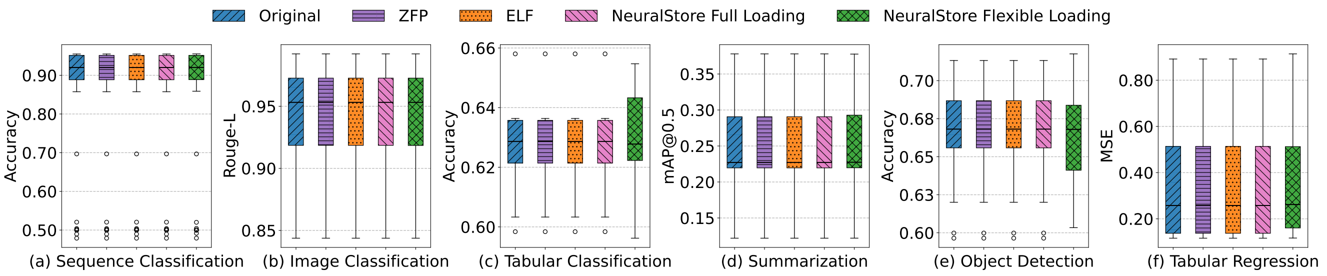


Figure 14: Model Performance of Compared Compression Algorithms under Different Tasks

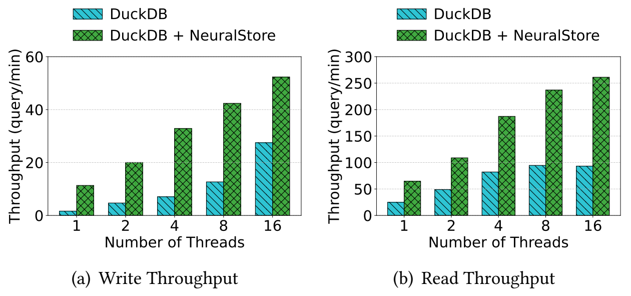


Figure 15: Performance of NeuralStore on DuckDB

experiments with two loading strategies, namely full loading and flexible loading with 8 bits.

First, we measure the throughput of the two strategies. The experiment is conducted by first initializing NeuralStore with 50 models using a precision tolerance of 5.96×10^{-8}. We then run 4 to 32 clients, each of which continuously sends the load model queries. Depending on the strategy, NeuralStore either loads full-bit-width or 8-bit delta tensors. The results are presented in Figure 12(a). It shows that 8-bit flexible loading achieves up to a 2.4× speedup compared to full loading. This improvement is attributed to the reduced number of bits that need to be read, which lowers both disk I/O and memory usage. Notably, the disk I/O becomes a bottleneck when models are concurrently retrieved by over 16 clients. In contrast, the flexible loading strategy avoids such a bottleneck by significantly reducing the bits loaded.

Second, we evaluate the number of bits saved for each delta tensor using the flexible loading strategy. To run the experiment, we initialize 800 models in NeuralStore, and load the models non-repeatedly. We record the number of bits saved for each tensor and display the results in Figure 12(b). It shows that flexible loading saves 8.4 bits on average, which indicates a compression ratio of 1.53×. With a precision tolerance of 5.96×10^{-8}, discarding this number of bits results in a precision loss of less than 10^{-4}. 5% of the delta tensors save 0 bits with flexible loading because their bit width is less than or equal to 8. These tensors mainly originate from the same base tensor as their corresponding delta tensors.

6.4.4 Performance Impact of Precision Tolerance. We then evaluate the impact of precision tolerance on model performance. We use the tasks and models as described in Section 6.1.1. For each task, we gradually increase the precision tolerance from 5.96×10^{-8} until a surge in models' average absolute performance change is observed. At each precision tolerance, we measure the average absolute accuracy change and compressed storage size. The results are shown in Figure 13. Across all tasks, increasing the precision tolerance leads to reduced storage consumption, as fewer bits are needed during quantization. However, the sensitivity to precision tolerance varies across tasks. For sequence classification, model performance change remains within 0.02% at tolerances below 1×10^{-5}. The model performance change begins to amplify from a tolerance of 2×10^{-3}, from where it increases from 0.42% to 3.22% at the tolerance of 2.8×10^{-3}. For image classification, model performance remains unaffected at tolerances below 1×10^{-5}, but starts increasing from 4.5×10^{-3}, reaching a peak change of 4.12% at 4.9×10^{-3}. Lastly, tabular classification models maintain a performance change below 0.6% until the tolerance reaches 2×10^{-2}, beyond which model performance change increases up to 3.91% at 6×10^{-2}. Users can configure a higher tolerance on a per-model basis to balance the trade-off between storage consumption and model performance degradation. For example, models for image classification tasks can safely use a precision tolerance up to 1×10^{-5}, since no performance change is observed in Figure 13 at this tolerance level.

Table 4: Summary of Models Used for Performance Change Evaluation

Domain	Task	Dataset	Architecture
NLP	Sequence Classification	IMDB	BERT (9) DistilBERT (22) RoBERTa (16)
	Summarization	SAMSum	T5-small (42) T5-base (20) BART-base (10)
CV	Image Classification	Stanford Dogs Beans Food-101 CIFAR-10	ViT-base (34) Swin-base (4) Swin-tiny (5)
	Object Detection	CPPE-5	DETR-ResNet-50 (18)
Tabular	Classification	Avazu	MLP (12)
	Regression	Regression	MLP (4) TabNet (4)

6.4.5 Performance Impact Across Models. We now evaluate the impact of flexible model loading and precision tolerance on model performance. In addition to the three tasks introduced in Section 6.1.1, we further include three more tasks to cover a wider range of model architectures: (1) summarization, where models generate abstractive summaries from dialogues, (2) object detection, where models identify multiple objects within images, and (3) tabular regression, where models predict continuous numeric values based on structured features. In total, our evaluation covers 200 models across six tasks. Due to the space constraints, we provide a summary of these models in the extended version [9]. We compress and decompress

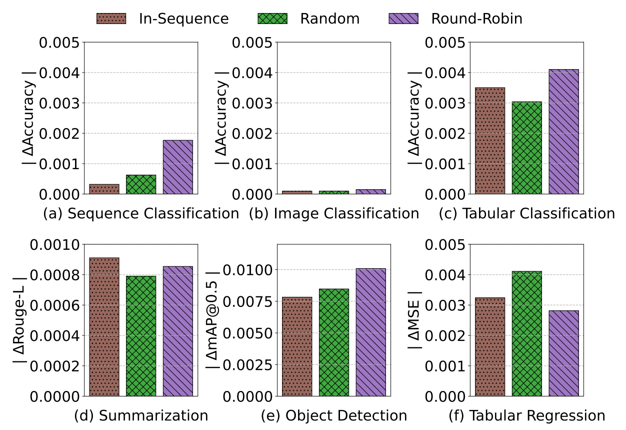


Figure 16: Model Performance Change under Different Orders

these models using a fixed precision tolerance of 5.96×10^{-8}, and measure their absolute performance change. The results are shown in Figure 14. For ZFP, ELF, and NeuralStore Full Loading, over 90% of models exhibit no performance change. For NeuralStore Flexible Loading, 57 out of 200 models show a performance change less than 0.01%, and over 70% of models exhibit a change within 0.1%. More than 95% of models remain within a performance change of 1%. The increase in performance change is expected, since flexible loading restores only the most significant 8 bits of each delta tensor. Among all tasks, object detection models are most sensitive to precision loss, with an average performance change of 0.8%. This is because the object detection task requires predicting bounding boxes in images, where small changes in model weights can lead to high deviations in predicted object locations. On the contrary, image classification models show the smallest performance change, with an average of 0.009%, due to the simplicity of the task.

6.4.6 Performance Impact of Storage Order. To assess the impact of different storage orders on model accuracy, we insert models in three different orders and evaluate their resulting performance changes. Specifically, we evaluate the following storage orders: 1) Random order, which is the default setting we use to avoid potential bias from fixed storage sequences; 2) Sequential order, where models fine-tuned from the same pre-trained model are stored consecutively, representing an optimal case; 3) Round-robin order, which alternates storage across different architectures to maximize randomness and complexity, representing a worst-case scenario. As shown in Figure 16, the insertion order can affect model performance. For the sequence classification, image classification, tabular classification, and object detection tasks, the round-robin order results in the highest model performance changes of 0.17%, 0.01%, 0.41%, and 1.00%, respectively. For tabular regression, the random order causes the highest change, while for summarization, the sequential order yields the highest change. Such performance deviations occur because tensors inserted earlier are more likely to be selected as bases in the similarity index. As a result, changing the insertion order can lead to different base-delta pairings during compression and thus slightly influence model performance. However, the overall effect of storing orders on model performance remains limited, as NeuralStore guarantees that the precision loss for each model stays within the user-defined tolerance.

6.5 Extensibility of NeuralStore

We now extend NeuralStore into DuckDB [34], denoted as DuckDB+NeuralStore, and assess its performance. For comparison, the baseline DuckDB applies Zstd compression to each serialized model before saving it. Both DuckDB+NeuralStore and baseline are configured with a 32GB memory limit. We vary the number of concurrent connection threads from 1 to 16 and measure write and read throughput. The results are shown in Figure 15. For write throughput, we can observe that both systems show performance gain as the number of concurrent threads increases. DuckDB+NeuralStore consistently outperforms the baseline across all levels of parallelism. It achieves a peak throughput of 52 queries per minute at 16 threads, which is 1.93× higher than that of the baseline. This is due to the fact that NeuralStore reduces the I/O cost by utilizing the proposed tensor pages and HNSW index caching. For read operations, NeuralStore also surpasses the DuckDB baseline across all levels of parallelism, achieving a peak throughput of 261 queries per minute, compared to 94 for the DuckDB baseline. These improvements are attributed to NeuralStore's in-memory index caching, on-demand decompression, and flexible model loading strategy. In addition, DuckDB+NeuralStore consumes only 78% of the storage used by the baseline DuckDB, demonstrating the overall effectiveness of our approach in reducing storage consumption.

7 Related Works

In-database Machine Learning. Recently, there has been a growing interest in in-database machine learning (ML), which aims to integrate model training and inference directly within database engines to minimize data movement and exploit database-native execution for scalable analytics. Early systems such as Bismarck [16], MADlib [20], and Oracle Machine Learning [6] embed learning algorithms into SQL-based workflows to enable large-scale model training over relational data. More recent efforts, such as InferDB [37], RAVEN [33], CorgiPile [42], Vertica-ML [15], push model inference into the database engine, optimizing the runtime serving path by tightly coupling inference with data access. In addition, systems like EVA [43] and VIVA [36] enable declarative definition of machine learning pipelines for in-database video analytics. NetsDB [50] proposes tensor deduplication during inference by identifying structural similarity across neural networks to improve inference efficiency. While these systems primarily focus on in-database ML pipelines, they offer limited support for model management. In contrast, NeuralStore enables efficient in-database model management, with a design tailored for modern DL models. We introduce a set of techniques all natively embedded in the DBMS engine, including a tensor-based storage engine, adaptive delta quantization, and compression-aware model loading, to bridge the gap between model storage and inference within DBMSs.

Model Management System. There are several existing dedicated model management systems. ModelDB [39] focuses on tracking model metadata, lineage, and experiment results to facilitate reproducibility and model governance, but it does not address model storage optimization. To reduce storage overhead, ModelHub [30] enables delta storage, which maintains the differences between fine-tuned and base models with explicit relations. However, ModelHub only captures pairwise differences between models and their predecessors, and does not account for the high entropy of floating-point weights, which limits its storage efficiency. In contrast, NeuralStore targets tensor-level deduplication across the entire model collection, while incorporating a delta quantization algorithm that can efficiently compress high-entropy delta tensors.

8 Conclusion

This paper introduced NeuralStore, an efficient in-database deep learning model management system. We introduced a tensor-based storage engine that enables fine-grained tensor deduplication by leveraging an enhanced HNSW-based tensor index. To further reduce storage costs while preserving model performance, we proposed an adaptive delta quantization algorithm that dynamically compresses delta tensors with bounded accuracy loss. Moreover, we designed a compression-aware loading and inference mechanism that supports direct computation on compressed tensors, significantly improving model retrieval and serving efficiency. Extensive experimental results demonstrate that, compared to state-of-the-art in-database model management systems, NeuralStore achieves substantial storage savings while maintaining competitive model retrieval throughput and inference accuracy.

References

[1] 2020. Beans Dataset.
[2] 2024. zlib. https://zlib.net.
[3] 2025. Avazu Dataset.
[4] 2025. Azure SQL. https://azure.microsoft.com.
[5] 2025. Hugging Face. https://huggingface.co.
[6] 2025. Oracle Machine Learning. https://docs.oracle.com/en/database/oracle/machine-learning.
[7] 2025. PostgresML. https://postgresml.org.
[8] 2025. Zstandard. https://github.com/facebook/zstd.
[9] Anonymous Author(s). 2025. *NeuralStore: Efficient In-database Deep Learning Model Management System (Extended Version)*. https://storage.googleapis.com/artifact_docs/p518.pdf
[10] Anonymous Author(s). 2025. *NeuralStore Implementation*. https://anonymous.4open.science/r/neurstore-80BD
[11] Christoph Brücke, Philipp Härtling, Rodrigo Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proc. VLDB Endow.* 16, 12 (2023), 3649–3661. https://doi.org/10.14778/3611540.3611554
[12] Deng Cai. 2021. A Revisit of Hashing Algorithms for Approximate Nearest Neighbor Search. *IEEE Trans. Knowl. Data Eng.* 33, 6 (2021), 2337–2348.
[13] Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, Yu Qiao, and Ping Luo. 2024. EfficientQAT: Efficient Quantization-Aware Training for Large Language Models. *CoRR* abs/2407.11062 (2024).
[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. *CoRR* abs/2401.08281 (2024).
[15] Arash Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *SIGMOD Conference*. 755–768.
[16] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD Conference*. 325–336.
[17] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.
[18] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630 (2021).
[19] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*.
[20] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.
[21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*.
[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.
[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *CVPR*. 2704–2713.
[24] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.
[25] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR* abs/1806.08342 (2018).
[26] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Trans. Vis. Comput. Graph.* 20, 12 (2014), 2674–2683.
[27] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. 2021. Post-Training Quantization for Vision Transformer. In *NeurIPS*. 28092–28103.
[28] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (Eds.). The Association for Computer Linguistics, 142–150. https://aclanthology.org/P11-1015/
[29] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.
[30] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. Towards Unified Data and Lifecycle Management for Deep Learning. In *ICDE*. 571–582.
[31] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or Down? Adaptive Rounding for Post-Training Quantization. In *ICML*, Vol. 119. 7197–7206.
[32] Beng Chin Ooi, Shaofeng Cai, Gang Chen, Yanyan Shen, Kian-Lee Tan, Yuncheng Wu, Xiaokui Xiao, Naili Xing, Cong Yue, Lingze Zeng, et al. 2024. NeurDB: an AI-powered autonomous data system. *Science China Information Sciences* 67, 10 (2024), 200901.
[33] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *SIGMOD Conference*. 587–601.
[34] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212
[35] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *KDD*. 1378–1388.
[36] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. 2022. Optimizing Video Analytics with Declarative Model Relationships. *Proc. VLDB Endow.* 16, 3 (2022), 447–460.
[37] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proc. VLDB Endow.* 17, 8 (2024), 1830–1842.
[38] Zhaoyuan Su, Ammar Ahmed, Zirui Wang, Ali Anwar, and Yue Cheng. 2024. Everything You Always Wanted to Know About Storage Compressibility of Pre-Trained ML Models but Were Afraid to Ask. *Proc. VLDB Endow.* 17, 8 (2024), 2036–2049.
[39] Manasi Vartak. 2017. MODELDB: A System for Machine Learning Model Management. In *CIDR*.
[40] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978.
[41] Naili Xing, Shaofeng Cai, Gang Chen, Zhaojing Luo, Beng Chin Ooi, and Jian Pei. 2024. Database Native Model Selection: Harnessing Deep Neural Networks in Database Systems. *Proc. VLDB Endow.* 17, 5 (2024), 1020–1033.
[42] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cédric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, Jieping Ye, and Ce Zhang. 2022. In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle. In *SIGMOD Conference*. 1286–1300.
[43] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *SIGMOD Conference*. 602–616.
[44] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference. In *MICRO*. 811–824.

[45] Lingze Zeng, Naili Xing, Shaofeng Cai, Gang Chen, Beng Chin Ooi, Jian Pei, and Yuncheng Wu. 2024. Powering In-Database Dynamic Model Slicing for Structured Data Analytics. *Proc. VLDB Endow.* 17, 13 (2024), 4813–4826.

[46] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2024. IMBridge: Impedance Mismatch Mitigation between Database Engine and Prediction Query Execution. In *SIGMOD Conference Companion*. ACM, 456–459.

[47] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2025. Mitigating the Impedance Mismatch between Prediction Query Execution and Database Engine. *Proc. ACM Manag. Data* 3, 3 (2025), 189:1–189:28.

[48] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. In *ICML*, Vol. 32. 838–846.

[49] Zhanhao Zhao, Shaofeng Cai, Haotian Gao, Hexiang Pan, Siqi Xiang, Naili Xing, Gang Chen, Beng Chin Ooi, Yanyan Shen, Yuncheng Wu, and Meihui Zhang. 2025. NeurDB: On the Design and Implementation of an AI-powered Autonomous Database. *CIDR* (2025).

[50] 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. *Proc. VLDB Endow.* 15, 10 (2022), 2230–2243.