ZipLLM: Efficient LLM Storage via Model-Aware Synergistic Data Deduplication and Compression

Zirui Wang[1], *Tingfeng Lan*[1], *Zhaoyuan Su*[1], *Juncheng Yang*[2], *Yue Cheng*[1]
[1]*University of Virginia,* [2]*Harvard University*

Abstract

Modern model hubs, such as Hugging Face, store tens of petabytes of LLMs, with fine-tuned variants vastly outnumbering base models and dominating storage consumption. Existing storage reduction techniques—such as deduplication and compression—are either LLM-oblivious or not compatible with each other, limiting data reduction effectiveness.

Our large-scale characterization study across all publicly available Hugging Face LLM repositories reveals several key insights: (1) fine-tuned models within the same family exhibit highly structured, sparse parameter differences suitable for delta compression; (2) bitwise similarity enables LLM family clustering; and (3) tensor-level deduplication is better aligned with model storage workloads, achieving high data reduction with low metadata overhead. Building on these insights, we design BitX, an effective, fast, lossless delta compression algorithm that compresses the XORed difference between fine-tuned and base LLMs. We build ZipLLM, a model storage reduction pipeline that unifies tensor-level deduplication and lossless BitX compression. By synergizing deduplication and compression around LLM family clustering, ZipLLM reduces model storage consumption by 54%, over 20% higher than state-of-the-art deduplication and compression approaches.

1 Introduction

Large language models (LLMs) have become foundational tools in modern artificial intelligence (AI). With the rapid progress in open-source LLM development [35, 45, 47–49], millions of LLMs are now publicly available through model hubs such as Hugging Face [21] and TensorFlow Hub [26]. These platforms support uploads, downloads, and sharing of base models and fine-tuned variants, enabling users to adapt models to diverse downstream tasks with minimal effort.

This trend has led to an explosion in the number of hosted models. As shown in Figure 1, Hugging Face alone hosts over 14 petabytes (PB) of models (as of Q1 2025), with storage volume growing exponentially, posing a serious threat to the sustainability of machine learning (ML) infrastructure.

Two *observations* underscore this challenge. First, *fine-tuned* LLMs vastly outnumber base models and contribute disproportionately to overall storage footprint, despite being only slight modifications. Second, LLM storage is dominated by two floating-point formats: `BF16` and `FP32`. While `FP32` is popular in terms of model count (often in smaller models such as those for computer vision), `BF16` accounts for the majority of total LLM storage size. These trends highlight the need to prioritize LLM-specific storage patterns in future optimizations.

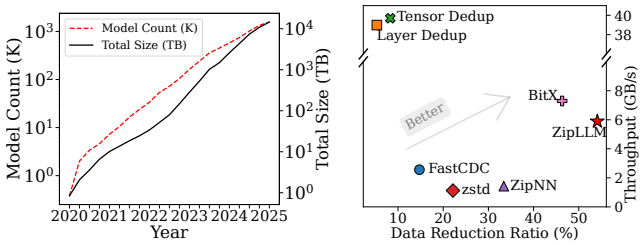


Figure 1: Left: Hugging Face's model count and storage consumption grow exponentially. Right: ZipLLM achieves both high data reduction and throughput. ZipLLM represents the end-to-end system throughput. while BitX shows the compression kernel throughput.

We collect *all* public LLM repositories from Hugging Face (Cutoff date March 2025) and conduct a *first-of-its-kind, large-scale study focusing on LLM storage*. Our analysis leads to the following *insights*:

- **Element-wise weight deltas are small and structured within LLM model families.** Fine-tuned models derived from the same base exhibit tiny differences, making them ideal for lossless delta compression.
- **Bitwise similarity enables LLM clustering and lineage tracking.** Bit distance, a new metric that we propose, based on the bitwise Hamming distance, serves as a lightweight, robust signal for identifying LLM families and potentially supporting applications like model provenance, duplicate detection, and clustering.
- **Chunk-based deduplication is LLM-oblivious and sub-optimal for modern model storage.** Chunk-level deduplication, such as content-defined chunking (CDC) [53, 56, 82], operates on raw byte streams without LLM structure awareness, resulting in the loss of crucial information needed for effective model-aware compression. It also scales poorly with storage capacity.
- **Model-aware, tensor-level deduplication is well-suited for LLM-aware lossless compressors**, offering reasonable data reduction, but with significantly higher performance and lower metadata overhead compared to CDC.

This paper makes the following contributions:

- We conduct a *comprehensive analysis* of Hugging Face's massive-scale model repository, with a focus on how LLM

Table 1: Comparison of model storage reduction techniques. Note that existing solutions are limited to use either deduplication **or** compression.

Solution	Compression	Deduplication	Cross-model	Throughput	Storage Reduction	Cons & Pros
HuggingFace Xet [79]	No	Yes	Yes	Low	High	No compression support
ELF [70]	Yes	No	No	High	High	Lossy compression
ZipNN [30]	Yes	No	No	Medium	Medium	Ignores cross-model redundancy
FM-Delta [58]	Yes	No	Yes	Low	Medium	Requires identical model structure; lacks `BF16` support
ZipLLM (ours)	Yes	Yes	Yes	High	High	Lossless and model structure-aware dedup and compression

family structure impacts storage redundancy and compression effectiveness.

- We introduce a *novel metric, bit distance*, to quantify the similarity between fine-tuned models and their base models.
- Building on this, we design BitX, a *highly effective, fast, lossless delta compression algorithm* that compresses LLM variants by encoding XOR-based deltas.
- We identify a *new ML system design principle*: for modern model storage systems, deduplication and lossless compression must be co-designed and unified to fully exploit model structure and redundancy.
- We build ZipLLM, a *model storage reduction pipeline* that synergizes tensor-level deduplication and lossless BitX compression, achieving higher storage savings for large-scale LLM repositories.

Evaluation results show that ZipLLM reduces the storage size of 3,048 sampled LLMs by 54.1%, 20% higher than the state-of-the-art methods. Meanwhile, ZipLLM achieves 2× higher compression throughput (Figure 1). Our implementation is publicly available at: https://github.com/ds2-lab/ZipLLM.

2 Background and Related Work

This section reviews existing techniques for reducing data storage, including both general-purpose and model-aware approaches. A high-level comparison of model storage compression and deduplication methods is summarized in Table 1.

2.1 Traditional Storage Reduction

General-purpose Compression. General-purpose data compression techniques, such as Zstandard (zstd) [11] and Brotli [5], are widely used in storage systems to reduce data size by exploiting local byte-level redundancy through methods like dictionary-based compression [77, 91] and entropy coding [16, 20, 32, 44, 62, 74]. While these methods are data-type agnostic, further compression gains can be achieved when the data type is known. For instance, run-length encoding and delta encoding are highly effective for low-entropy [69] and file changes [43, 73], and specialized techniques have been developed for columnar and time-series datasets [9, 10, 36, 37, 41, 75].

Lossy compression methods—such as ZFP [18, 40] and SZ [17, 38, 71]—though effective in scientific domains, are unsuitable for model storage due to their inability to guarantee exact recovery. Similarly, while quantization is a popular lossy compression approach for model inference [15, 22, 39, 83, 87, 88], it is a user-driven choice and orthogonal to storage system design. For model storage, lossless compression that preserves floating-point precision remains essential for correctness.

General-purpose Deduplication. Deduplication is a widely used technique for reducing storage footprint by identifying and storing only unique data blocks, replacing duplicates with references. File-level deduplication, used in systems like Git LFS [25], eliminates exact file copies with minimal metadata overhead but cannot detect partial redundancy. Chunk-level deduplication, especially content-defined chunking (CDC) [53, 59, 82], addresses this by splitting files into variable-sized chunks based on content, enabling more effective duplicate detection despite insertions or shifts. CDC has been adopted in production systems such as NetApp ONTAP [54, 55] and Dell EMC Data Domain [13, 14]. However, CDC is ill-suited for model storage due to high metadata overhead from massive variable-sized chunks, making it impractical for processing large-scale model storage efficiently.

2.2 Model-aware Storage Reduction

Model-aware Compression. State-of-the-art model-aware compression methods exploit model parameter distribution patterns and focus on the IEEE 754 floating-point format [2], which represents each number using a *sign bit*, an *exponent*, and a *mantissa*. By exploiting the fields with high redundancy, different compression strategies are proposed to reduce model storage space [30, 58, 70].

ELF [70] eliminates exponent bits by mapping weights into a normalized range, but it is inherently lossy and unsuitable for model hubs that require exact recovery. ZipNN [30] improves compressibility by reordering float bytes to isolate compressible fields like sign and exponent bits. Although it could exploit cross-model redundancy, the released implementation only supports files with identical sizes, preventing compression when layer dimensions differ (e.g., modified embedding layers). FM-Delta [58] targets cross-model redundancy by computing weight deltas between fine-tuned and base models. However, it requires strict architectural alignment, lacks `BF16` support, and achieves low throughput (around 100 MB/s), limiting its applicability at scale. DeltaZip [86] and BitDelta [42] are two recently proposed techniques that also explore cross-model redundancy. However, both methods are lossy: they approximate or quantize weight differences to reduce GPU memory cost, making them unsuitable for model hubs that require exact recovery of model weights.

Despite being model-aware, ELF, ZipNN, FM-Delta, DeltaZip, and BitDelta require intact model structures (e.g., aligned tensors and parameters) to function effectively. How-

ever, once a model is partially deduplicated at the chunk level (e.g., via CDC [53]), the remaining unmatched regions lose their structural boundaries and appear as fragmented byte sequences. As a result, these residual parts can no longer be processed by compressors that require intact model structure, preventing them from exploiting redundancy.

Model-aware Deduplication. Hugging Face employs a two-stage deduplication strategy that combines file-level and chunk-level techniques, including CDC [79]. File-level deduplication removes exact duplicates by comparing content hashes, which is effective for detecting re-uploaded files. To address inefficiencies in handling large files with minor changes—common in fine-tuning and checkpointing—CDC divides files into variable-sized chunks using a rolling hash, enabling only modified chunks to be uploaded. This chunk-based approach, backed by content-addressed storage (CAS), significantly reduces redundancy. Their early findings report up to a 50% reduction in storage usage and improved upload and download speeds compared to Git LFS [80].

3 Characterizing Hugging Face Model Storage

The number of public LLM repositories has grown rapidly in recent years, driven by the popularity of open-source model families (e.g., Llama [27,89], Mistral [35]) and the widespread practice of fine-tuning base models for domain-specific tasks. This proliferation is further fueled by community sharing and continuous model versioning. As a result, the demand for scalable and efficient storage systems has significantly increased. Public model hubs such as Hugging Face [21] and TensorFlow Hub [26] now host millions of LLMs, with the number growing exponentially. This exploding growth places substantial demands on backend storage systems, both in terms of capacity and bandwidth. To better understand this landscape, we characterize *all* publicly accessible LLMs hosted by Hugging Face, the world's largest model hub.

3.1 Model Storage Explosion

We define LLM model storage as the storage of parameter files associated with LLMs. Compared to auxiliary files such as configuration files or tokenizers, parameter files dominate the overall storage footprint. These files are typically stored using cloud-based object storage services [4, 6, 31], which provide efficient access and scalability.

To quantify this growth, we examine statistics from Hugging Face. As shown in Figure 1 (left), the number of public models (including LLMs and non-LLM models) on Hugging Face has surpassed 1.5 million in 2025, up from 500K just one year earlier. The storage footprint grew nearly 6× during the same period, exceeding 14 PB for public models (excluding private ones) in early 2025. Public models are just the tip of the iceberg—a significant portion of models hosted on the platform are private repositories, which are inaccessible to the public. Our observation corroborates with a recent Hugging Face blog [79].

Implications

This exponential growth trajectory continues to place mounting pressure on model hubs. Looking ahead, we project that by the end of 2025, model storage demand will continue its exponential trajectory. This data explosion poses a grand challenge to the long-term sustainability of ML/AI infrastructure.

3.2 Model Storage Format

The shift in model storage formats has significant implications for deduplication and compression. As shown in Figure 2a, `safetensors` [34] and `GGUF` [24] have become the dominant formats on Hugging Face, together accounting for over 90% of total storage in 2025.

Safetensors Format. `Safetensors` introduces structured metadata and consistent tensor layouts, which enable model-aware compression and deduplication. Unlike legacy formats like `.bin` or `.h5` [1], which often serialize data with model-specific headers and variable alignment, Safetensors format is also zero-copy and metadata-aware, supporting parallel access to individual tensors without scanning the full file. This enables high-throughput model loading.

GGUF Format. `GGUF` is a lightweight, extensible format for storing *quantized* models, featuring structured metadata for modern tooling compatibility. It addresses limitations of earlier formats and has become the standard for quantized LLMs.

Implications

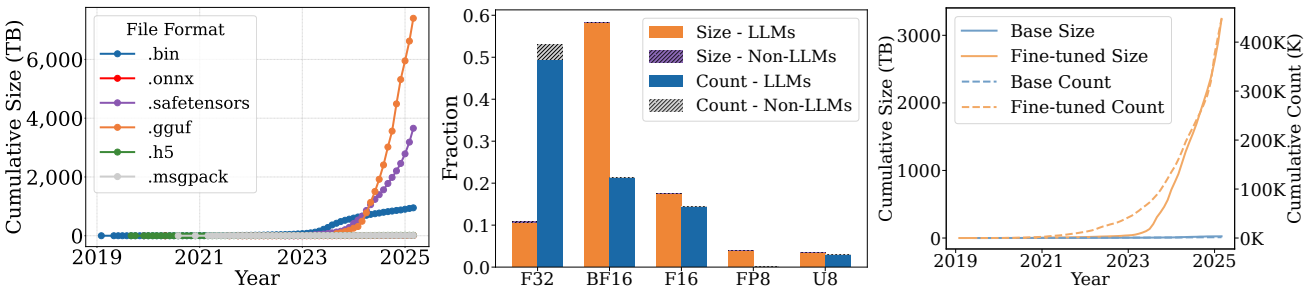
*These two formats reflect the **dual** nature of modern LLM workflows: high-precision models are used for training and fine-tuning, while quantized models are used for resource-efficient inference. This dichotomy presents distinct storage patterns and optimization opportunities. As a result, our compression and deduplication techniques are specifically designed to target these formats, which are central to both current and future usage of model storage systems.*

3.3 Data Type Distribution

To better understand which floating-point types contribute most to model storage, we analyze the distribution of data types across models. As shown in Figure 2b, `BF16` is the dominant format in terms of storage consumption, while `FP32` is the most common in terms of model count.

This discrepancy arises because many models use mixed precision: a few layers in `FP32` while the majority are `BF16` [1], or they are small non-LLM models (e.g., CV or traditional NLP models) that used `FP32`. As a result, while `FP32` appears in many models, these models are often small in size. In contrast, `BF16` is the standard format for large LLM checkpoints, contributing to its substantial share of total storage.

[1] In our analysis, if a model contains multiple data types (e.g., both `FP16` and `FP32`), it is counted in each corresponding category.



(a) Cumulative model storage by file format. **(b)** Top data types by size and model count. **(c)** Growth of base and fine-tuned models.

Figure 2: Measurement of model repositories on Hugging Face. Note that non-LLMs contribute to a tiny fraction of storage consumption.

Given these findings, we focus our compression experiments on `BF16` and `FP32`, as they are the most prevalent formats in modern repositories. Both use 8-bit exponents, which simplifies our system design and allows unified handling of their binary representations. While we focus our evaluation on `BF16`, our compression techniques are *data-type-agnostic* and can be generalized to other formats.

Implications

The vast majority of storage consumption comes from LLMs, particularly those using the `BF16` format. Non-LLMs contribute minimally in size. As such, future storage optimizations should prioritize LLM-specific formats and patterns to maximize storage efficiency.

3.4 LLM Families

3.4.1 Base and Fine-tuned LLMs

Beyond data types, another key factor influencing storage footprint is model lineage [57, 90]. It is well known that the majority of today's LLMs are derived from a very small set of base models [29, 52]. This observation motivates our next analysis, which focuses on LLM families—groups of fine-tuned models that share a common base. Understanding this structure is essential for identifying *approximate redundancy*[2] and guiding targeted storage reduction strategies.

Figure 2c shows the cumulative count and storage size of base models and fine-tuned models over time. We observe that fine-tuned models have rapidly outpaced base models in both quantity and storage footprint. As of early 2025, fine-tuned models account for at least 3.2 PB of data, representing 99.22% of the total storage (3,243.17 TB out of 3,268.72 TB). In terms of count, there are 447,457 fine-tuned models, comprising 99.64% of all models. Notably, this is a conservative estimate, as some fine-tuned models lack proper metadata (e.g., model cards) and thus cannot be reliably identified.

This trend suggests that optimizing the storage of fine-tuned models is crucial to improving the overall efficiency of model repositories.

[2]Defined as hidden, structural redundancy not directly removable via exact deduplication.

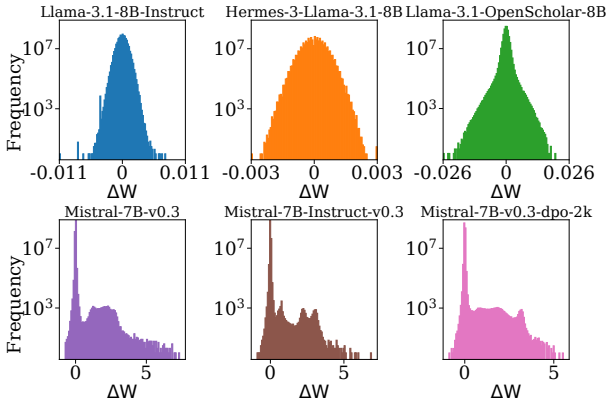


Figure 3: Distribution of element-wise weight differences against `Llama-3.1-8B`. Top row: Deltas for three models fine-tuned directly from `Llama-3.1-8B`. Bottom row: Deltas for three models from the `Mistral-7B-v0.3` family.

3.4.2 Cross-model Parameter Difference

Since most fine-tuned LLMs share the same model structure with base models [27, 89]—meaning each tensor shares the same shape and position—a natural and direct approach is to analyze the element-wise differences in their weights (model parameters). To verify that such similarity is indeed prevalent, we compute the value differences (delta Δw) at each parameter position i as $\Delta w_i = w_i - \hat{w}_i$, where w_i and $\hat{w}_i$ represent the i^{th} float value in the fine-tuned and base model, respectively. Here, the index i corresponds to the position of each float in the serialized model file, obtained by traversing all tensors in their original storage order and flattening each tensor in row-major layout. This delta is computed across all tensors to capture fine-grained numerical changes between models.

We begin with the `Llama-3.1-8B` [47] base model and select three of its fine-tuned variants. As shown in Figure 3 (top), the delta values are small and centered around zero, with similar bell-shaped distributions across all variants. This indicates that *fine-tuned models in the same family introduce only minor modifications to the base model's weights.*

To validate whether this property holds across model families, we repeat the same analysis using models from the `Mistral-7B-v0.3` [35] family. See Figure 3 (bottom). Although the architectures are almost identical (except for the

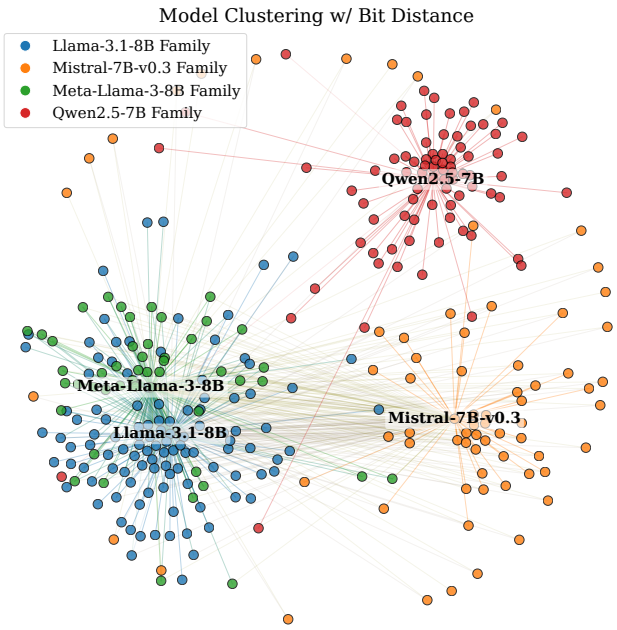


Figure 4: Clustering of 311 LLMs by bit distance.

embedding and `lm_head` layers), the resulting delta distributions are much wider and and asymmetric. The element-wise differences are significantly larger, suggesting that models from different origins have less similarity—even if their architecture matches. We tested many models and found consistent results, which are omitted due to space limits.

Implications

Element-wise weight deltas can serve as a simple, efficient, and robust tool for identifying model lineage and clustering models by family. Fine-tuned variants derived from the same base model consistently exhibit small and structured deltas, making them well-suited for delta compression.

3.4.3 Cross-model Bit Distance and LLM Clustering

Building on the observed correlation between element-wise deltas and model similarity, we propose a bitwise distance metric that measures how many bits differ between two model files. Given two models with the same architecture, we align their floating-point weights in original order and compute the bit distance as follows:

$$\texttt{Bit Distance:} \quad \mathcal{D}(\mathbf{w}, \hat{\mathbf{w}}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{H}(w_i, \hat{w}_i) \qquad (1)$$

Here, n is the total number of float values in the models. w_i and $\hat{w}_i$ denote the ith float value from the model pair $\mathbf{w}$ and $\hat{\mathbf{w}}$, respectively, both represented in raw binary format. $\mathcal{H}(w_i, \hat{w}_i)$ computes the number of differing bits (i.e., the Hamming distance [28]) between the two binary representations. The final bit distance measures the average number of differing bits per float across the two models.

Using this metric, we compute pairwise distances across 311 models from four major LLM families: `Llama-3.1` [47], `Llama-3` [27], `Mistral` [35], and `Qwen2.5` [85]. We then construct a similarity graph by connecting model pairs with bit distance below a fixed threshold in Figure 4. We observe clear clustering behavior: models within the same family tend to form dense groups, while connections across families are sparse. This supports our earlier hypothesis and findings—models that share a pretrained origin exhibit high structural redundancy, even at the bit level. In contrast, different pretrained base models or models fine-tuned from different bases diverge significantly in their binary representation.

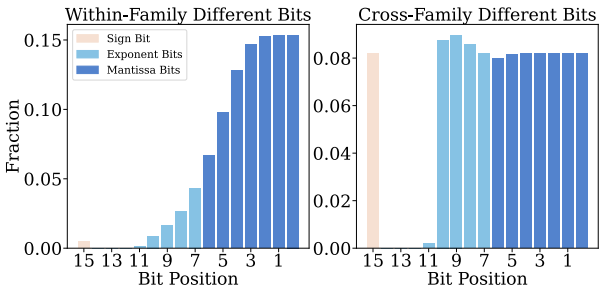


Figure 5: Bitwise contribution breakdown for bit distance. Left: Bit-level differences between a fine-tuned model and its base model within the same LLM family. All models are of `BF16`. Right: Differences across models from different families. The Y-axis indicates the fraction of total differing bits at each bit position, computed by dividing the number of bitwise XOR results with a 1 at that position by the total number of 1s across all 16 bits.

To better understand which bits contribute most to the observed bit-level differences between models, we break down the bit distance by position within the 16-bit `BF16` format, as shown in Figure 5. We observe that within the same family, most differences are concentrated in the lower mantissa bits, with the upper mantissa and exponent bits contributing far less, and the sign bit almost never flipping. This indicates a high degree of bit-level similarity, particularly in the high-order bits, which provides a good compression opportunity. In contrast, cross-family comparisons exhibit nearly uniform bit differences across all bit positions, with the exception of a few exponent bits (typically 1–2), which show slightly lower divergence. It reflects their much lower alignment and compatibility for compression. These findings further support the utility of LLM-family-aware compression techniques.

Implications

Bit-level similarity provides a powerful signal for organizing model repositories and guiding LLM storage optimizations. Models that are close in bit distance are more likely to benefit from delta encoding [63, 78], XOR-based compression [61], or structural reuse [37].
Beyond compression, the bit distance metric offers broader implications for large-scale model hubs such as Hugging Face, where accurate and automated identification of model lineage is missing and remains a challenge. Current tools often rely on manually curated metadata. In contrast, bit dis-

Table 2: FileDedup stats of Hugging Face model repositories.

Metric	Value
Total files	5,688,779
Duplicate files	1,182,818
Total size	11.89 PB
Saved size	0.97 PB (8.2%)
Repos with files can be deduped	506,337 (33.2%)

tance enables content-based provenance analysis, opening the door to a range of applications such as lineage tracking [57], duplicate detection [80], model clustering [3], and even LLM testing and evaluation [90].

3.5 Storage Redundancy in LLMs

Data deduplication is a widely adopted technique in large-scale storage systems to reduce storage costs by eliminating redundant data [23, 51, 60, 68, 76]. The effectiveness of deduplication depends heavily on the granularity at which it is applied. As mentioned in §2, file-level deduplication (FileDedup) offers low overhead and high throughput, but achieves only limited storage savings. In contrast, chunk-level deduplication (ChunkDedup) analyzes data at a finer granularity, enabling greater storage reduction at the cost of significantly higher metadata storage.

3.5.1 File-level Deduplication

To understand the redundancy landscape in model repositories, we first analyze FileDedup across all hosted Hugging Face models. FileDedup identifies redundancy by computing cryptographic hashes of entire model files and eliminating duplicates with matching fingerprints. As shown in Table 2, out of 5.6 million model files, approximately 1.18 million files are exact duplicates. FileDedup can eliminate these duplicates, reducing the total storage footprint by nearly 1 PB. Notably, more than 500,000 repositories—roughly one-third of all currently hosted—contain at least one redundant file, often due to users re-uploading unmodified model artifacts.

3.5.2 Deduplication with Content-defined Chunking

While FileDedup captures exact matches, we observe significant *partial* redundancy between model files that differ slightly, such as checkpoints from the same training run or fine-tuned variants. Deduplication at a finer granularity may help. To this end, we inspect the content of redundant chunks identified by ChunkDedup. We find that most deduplicated chunks correspond to serialized tensor data—indicating that the effectiveness of content-defined chunking (CDC) [56, 82] is largely due to repeated tensors[3] across *related* models, rather than generic byte-level similarity. This observation reveals that *although CDC can detect sub-file redundancy, the underlying source of duplication is often a tensor.*

[3] An LLM file may contain multiple layers, each with one or more tensors.

Implications

CDC is completely LLM-oblivious and operates directly on raw byte streams. While widely used in industry, it suffers from poor parallelizability and high metadata overhead, limiting its practicality for large-scale model repositories (see §5.3.1). By contrast, operating directly at the tensor granularity—where structure is explicitly defined—can achieve similar deduplication ratios but is naturally more parallelizable.

4 Design

This section presents ZipLLM, an LLM-aware storage reduction pipeline designed to eliminate redundancy in LLM model hubs efficiently. ZipLLM combines two complementary strategies—data deduplication and lossless compression—in an LLM-aware manner. These two techniques target different forms of redundancy and operate on separate dimensions, so they do not interfere with each other, but together maximize storage reduction. Unlike conventional CDC-based approaches that operate on raw byte streams, ZipLLM performs deduplication directly at the tensor level, leveraging model structure explicitly exposed in LLM formats. This design not only improves efficiency and deduplication ratio but also preserves the tensor structure required by downstream LLM-aware compressors (e.g., BitX). Built on these principles, ZipLLM eliminates exact redundancy through FileDedup and tensor-level deduplication using TensorDedup, and reduces approximate redundancy via BitX compression—all while preserving losslessness. We begin by describing each component of ZipLLM in detail and then explain how they work together in the end-to-end data reduction pipeline.

4.1 Model-aware, Tensor-level Deduplication

Our earlier analysis from §3.5.2 reveals that most repeated chunks correspond to serialized tensors, rather than arbitrary byte patterns. This suggests that the underlying source of duplication is *structural* in nature—driven by repeated tensors across fine-tuned or checkpointed models. Instead of relying on expensive compute to find duplicated chunks from long byte streams, ZipLLM leverages model semantics and performs deduplication directly at the tensor level.

Modern model formats (`safetensors` and `GGUF`) are naturally aligned with TensorDedup; they both store model weights in a structured format containing a header followed by serialized tensors. The header contains metadata describing each tensor, including name, shape, data type, and byte offset within the file. By parsing the header first, ZipLLM can efficiently locate each tensor, enabling parallel processing.

While we describe deduplication as part of ZipLLM, it can also be implemented as part of client applications, such as Git LFS [25]. When integrated into the client, TensorDedup avoids uploading redundant data to the storage server with-

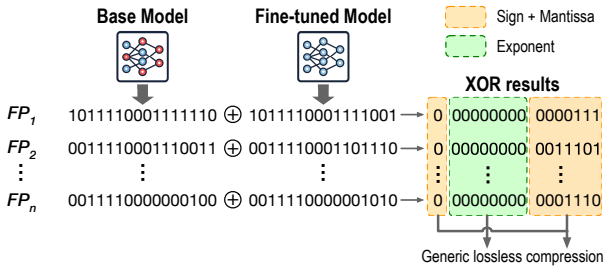


Figure 6: The BitX compression workflow. The example uses `BF16`, but BitX can support all floating-point types.

out excessive communication[4]. This can significantly reduce model upload time and network transfer for users.

4.2 BitX Delta Compression

Our earlier analysis from §3.4 reveals a key structural property of modern LLMs: fine-tuned models within the same LLM family exhibit small, consistent differences from their base models, both at the parameter value level (Figure 3) and at the bit level (Figure 4). These differences are often localized and sparse, forming a strong basis for compression. In particular, element-wise deltas show that most parameters remain nearly unchanged during fine-tuning, while the bit-level similarity confirms that related models can be clustered based on shared pretrained origin. Together, these insights motivate a compression strategy that directly encodes the bitwise differences between models. This is the foundation of our Bit-XOR (BitX) approach, which exploits fine-grained redundancy for efficient and lossless model storage reduction.

BitX Workflow. §3.4.3 shows that models in the same family often exhibit significant bit-level similarity. ZipLLM introduces a new compression algorithm, BitX, which exploits this bit-level redundancy to reduce storage consumption. Figure 6 illustrates the BitX workflow. Given a base model and a fine-tuned model that share the same architecture, BitX first aligns all floating-point values in their original order. For each corresponding pair of floats, BitX performs a bitwise XOR operation. This generates a sequence of XOR results, where many bits—especially in the sign, exponent, and high mantissa bits—are expected to be zero due to high redundancy. The XOR results capture the fine-grained differences between the models. Since most of the XOR bits are zero, the resulting sequence is highly compressible. BitX then applies a generic lossless compression algorithm, such as zstd [11], to further reduce storage. This two-stage process efficiently eliminates redundancy by directly encoding only the minimal changes required to reconstruct the fine-tuned model from its base.

Why XOR? We choose XOR rather than numerical differencing because it generates more zero bits at the binary level, leading to higher compressibility. For two similar floating-point numbers, numerical differencing often yields a new value with different exponents and mantissas, making the output denser and harder to compress. In contrast, XOR preserves bit-level similarity in the exponent and mantissa, producing sparse outputs that are far more amenable for compression.

By focusing on the bit-level delta between aligned floats, BitX achieves much higher compression ratios (§5.2) for fine-tuned models than traditional methods, without sacrificing accuracy or requiring any changes to model architectures.

4.3 LLM Clustering Thresholding

One key feature of the BitX workflow is family-based compression, which relies on explicit and accurate family clustering. Missing or inaccurate family information (e.g., pre-labeled tags) can significantly degrade compressibility. To robustly identify LLM models within the same family without pre-labeled metadata, we utilize the proposed bit distance metric (§3.4.3) to classify whether two models belong to the same family (within-family) or not (cross-family). First, models with different architectures or tensor shapes can be quickly categorized as cross-family models. For models with the same tensor shapes—which are much harder to distinguish—the decision is made by comparing the pairwise bit distance to a threshold: pairs with distances below the threshold are classified as within-family. In practice, the number of such comparisons can often be reduced to fewer than five, depending on the number of relatively similar variants of base model (e.g., `Llama-3`, `Llama-3.1`, `Llama-3.2`). This section presents our numerical method for determining the clustering threshold.

Following observations from previous works [19, 58], we assume the parameter weights $\mathbf{w} \sim \mathcal{N}(0, \sigma_w^2)$ and their fine-tuning deviations $\Delta\mathbf{w} \sim \mathcal{N}(0, \sigma_\Delta^2)$ follow symmetric Gaussian distributions centered at 0. Given definition (1), we define the expected bit distance between the base weights $\mathbf{w}$ and the fine-tuned weights $\hat{\mathbf{w}} = \mathbf{w} + \Delta\mathbf{w}$ as follows:

$$\mathbb{E}[\mathcal{D}(\mathbf{w}, \hat{\mathbf{w}})] = \iint \mathcal{D}(\mathbf{w}, \mathbf{w}+\delta)\, p_{\Delta\mathbf{w}}(\delta)\, p_{\mathbf{w}}(\mathbf{w})\, d\delta\, d\mathbf{w}$$

where δ denotes the perturbation added during fine-tuning, i.e., $\Delta\mathbf{w}$, and $p_{\mathbf{w}}(\cdot)$, $p_{\Delta\mathbf{w}}(\cdot)$ represent the probability density functions of the base weights and their perturbations.

However, the bit distance function $\mathcal{D}(\mathbf{w}, \hat{\mathbf{w}})$ is not continuous. Even small changes in floating-point value can cause sudden bit flips in the bit representation when the $\Delta\mathbf{w}$ cross the ULP (Unit in the Last Place) boundaries, i.e., the smallest spacing between two adjacent representable numbers. For example, changing a value from 1.000 to 1.001 may result in 5 flipped bits in its IEEE 754 representation, even though the numerical change is very small. Because of this, it is challenging to compute the expectation using analytical methods. To address this, we adopt a Monte Carlo approach [50] to estimate the expected range of bit distance, by sampling from the assumed distributions of the base weights and perturbations:

$$\hat{\mathbb{E}}[\mathcal{D}(\mathbf{w}, \hat{\mathbf{w}})] = \frac{1}{N} \sum_{i=1}^{N} \mathcal{D}(\mathbf{w}^{(i)}, \mathbf{w}^{(i)} + \delta^{(i)})$$

[4] ChunkDedup is typically performed on storage servers, requiring users to upload full data, as it needs orders of magnitude more hash comparisons.

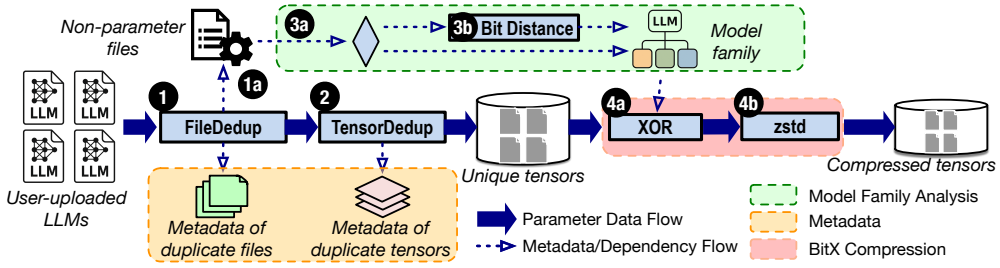


Figure 7: Overview of the ZipLLM storage reduction workflow.

where each $\mathbf{w}^{(i)} \sim \mathcal{N}(0, \sigma_w^2)$ and $\delta^{(i)} \sim \mathcal{N}(0, \sigma_\Delta^2)$. Here, N denotes the number of Monte Carlo samples used to approximate the expectation. We set $N = 100{,}000$ to ensure a stable estimation while keeping the computation efficient.

Empirically, for within-family fine-tuned models, we observe that the base model parameters have a standard deviation in the range of $\sigma_w \in [0.015, 0.05]$, and the fine-tuning deviations lie within $\sigma_\Delta \in [0.00, 0.02]$. The resulting expected bit distance values are consistently within the range of $[3.5, 6]$.

In contrast, cross-family model pairs show bit distance exceeding 6, due to larger weights differences across families as shown in Figure 3. Notably, for closely related model iterations such as `Llama-3` and `Llama-3.1`, the bit distance remains below 6—around 4. Based on these findings, we set a threshold of 4, which yields a classification accuracy of 93.5% for predicting whether two models belong to the same family. We present detailed sensitivity analysis in §A.1.

4.4 Putting It All Together: ZipLLM Design

Figure 7 illustrates the overall design of our LLM storage reduction pipeline, ZipLLM, which is tailored to the unique data characteristics of LLM storage.

In Step ❶, ZipLLM deduplicates files by computing content hashes and removing exact duplicates. In Step ❷, ZipLLM extracts all tensors across repositories and hash them individually to identify repeated tensors. These unique tensors are stored in a global tensor pool. ZipLLM also extracts metadata such as model cards [33] from non-parameter files (Step ❶a), and uses them to group models into families (Step ❸a). When the model family metadata is missing or incomplete, ZipLLM uses bit distance for similarity search (Step ❸b) to identify the closest base model (see §3.4.3). In Step ❹, ZipLLM performs BitX compression, which consists of two sub-steps. In Step ❹a, XOR deltas are computed between fine-tuned tensors and their corresponding base tensors, producing sparse binary differences. In Step ❹b, these XOR results are further compressed using generic algorithms such as zstd, yielding the final compact representation.

4.4.1 File-level Deduplication

ZipLLM first performs FileDedup on uploaded LLMs. This deduplication mechanism is particularly effective for exact reuse cases. For example, we observe that many users upload copies of popular base models (e.g., `Llama-2-7B`, `Mistral-7B`) without any modification. These files can be deduplicated entirely without decoding or parsing their contents. In addition, FileDedup is also used as a *prefiltering step* for downstream compression. If a file is unique based on its hash, ZipLLM proceeds with TensorDedup and subsequent compression. If the file is a duplicate, it is simply linked to a previously stored reference to avoid redundant storage.

4.4.2 Tensor-level Deduplication

While FileDedup is effective for detecting exact file reuse, it cannot capture partial redundancy within or across files. To address this, ZipLLM performs TensorDedup by directly operating on tensors stored in model files. For each model file, ZipLLM extracts all individual tensors and compute a hash for each. These hashes are then used to identify duplicate tensors across the entire corpus—within the same file, across multiple files in an LLM repository, or even across different repositories. All *unique* tensors are stored in a global *tensor pool* storage to enable reuse and eliminate redundant storage.

TensorDedup eliminates the inefficiencies of chunk-based deduplication, which suffers from high metadata overhead and limited semantic awareness. Compared to layer-level deduplication (LayerDedup), which treats an entire layer as one unit, TensorDedup offers finer granularity and better tolerance to minor changes in individual layers. We compare these deduplication methods later in §5.3.1 with detailed statistics in Table 5.

4.4.3 Lossless Compression

After deduplication, ZipLLM performs LLM-family-aware BitX compression across fine-tuned and base models.

Model Lineage Extraction. This step analyzes the configuration and metadata files extracted from non-parameter files (e.g., `config.json`, `README.md`) to identify lineage relationships among models. We use a combination of regular expressions and an LLM-based parser to extract base model information. Specifically, we parse architectures, tokenizers, and family identifiers to group structurally similar models.

Bit Distance Matching. If the metadata is missing or incomplete—for example, when the model card only specifies a general base model category (e.g., `Llama`) without naming a specific base model—ZipLLM identifies (multiple) likely base models using the structural information. ZipLLM then computes pairwise bit distances between the fine-tuned model

and candidate base models with matching shape and data type (§3.4.3). The model with the smallest bit distance is selected as the inferred LLM family.

Compressing XOR-ed Tensors. Once a base model is chosen, ZipLLM XORs aligned tensors from the fine-tuned and base models. This produces a sparse binary delta, which is lossless while highly compressible. The generated delta will then be compressed by a general-purpose compressor. By exploiting the approximate redundancy between base and fine-tuned models within the same family, ZipLLM applies generic compression methods (e.g., zstd) on the resulting deltas, achieving significant storage reduction for vast fine-tuned LLM corpora.

4.4.4 LLM Serving

To support efficient LLM loading and serving, ZipLLM stores minimal metadata alongside compressed model files. For each model, we record its associated base model, the hash of each tensor, the byte offset of each tensor in the original file, and the original `safetensors` metadata header. During compression, ZipLLM additionally stores the base model's tensor hashes used for BitX. At decompression time, ZipLLM first locates and decompresses each tensor. If a base tensor hash is present, ZipLLM retrieves the corresponding base tensor and apply XOR to reconstruct the original tensor. All tensors are then reassembled with the metadata header and written in parallel to produce a fully restored model file.

Fallback Strategy. ZipLLM is designed to be robust even when reference base models are missing. On large model hubs, it is common for multiple copies of the same base model to exist; if one copy is unavailable, ZipLLM substitutes an equivalent version. In the rare case where all original base models are removed, ZipLLM selects the most similar fine-tuned model (measured by bit distance) as a surrogate base and computes an additional XOR mask to account for differences. Applying this mask during decompression guarantees exact reconstruction of the target model. For additional robustness, ZipLLM compares this surrogate-based approach against standalone ZipNN compression and automatically selects the option yielding the better compression ratio.

5 Evaluation

5.1 Experimental Setup

Dataset. Because of the scale of Hugging Face (tens of PB LLMs stored), we randomly sampled 3,048 open-source LLM repositories from Hugging Face. Our dataset consumes 43.19 TB in raw size (Table 3). These repositories span a diverse range of model architectures, including 968 models derived from `Qwen2.5` [85], 151 from `Qwen3` [84], 139 from `Mistral` [35], 114 from `Llama-3` [45], 1,431 from `Llama-3.1` [46], 47 from `Llama-3.2` [48], 135 from `Gemma-2` [67], and 63 from `Gemma-3` [72].

Table 3: Model statistics summary.

Metric	Value
Model count	3,048
Total size	43.19 TB
Size after file dedup	41.80 TB

We exclude LoRA-only repositories from our evaluation. The reason is twofold: (i) LoRA adapters are highly heterogeneous in structure, making them difficult to find a base; and (ii) their sizes are negligible compared to corresponding base models, typically around 1%. Therefore, after sampling model repositories, we filtered out those that only contain LoRA adapters, leaving us with 3,048 full fine-tuned model repositories. For PEFT-style models, ZipLLM by default applies ZipNN to compress the adapters.

Baselines. We compare ZipLLM with both real-world production systems and recent state-of-the-art algorithms:

- **FileDedup and ChunkDedup (FastCDC)** are used by Hugging Face [80]. Because model information is lost during ChunkDedup, Hugging Face does not use compression in conjunction with the deduplication.
- **ZipNN** is the state-of-the-art model compressor that groups float numbers' different components for compression [30]. Because it does not consider deduplication, we added FileDedup to ZipNN for a fair comparison.
- **Compress-then-FastCDC** is a baseline we design to study the effect of execution order. In this setting, we first apply a compression algorithm (e.g., zstd) and then perform ChunkDedup (FastCDC). This allows us to evaluate how the ordering between compression and deduplication impacts the overall reduction efficiency.

Note that we did not compare with FM-Delta [58] and ELF [70], because FM-Delta does not support `BF16`, which is the most popular data type for LLMs, and ELF is lossy.

At the component level, we compare deduplication across different granularities (FileDedup, LayerDedup, ChunkDedup, and TensorDedup), as well as zstd compression algorithm.

Implementation. We implemented ZipLLM entirely in Rust. In total, our implementation comprises over 6,000 lines of code. For ZipNN and FastCDC, we use the open-source repos from the authors.

Metrics. We evaluate system efficacy using multiple metrics.

- **Data reduction ratio** calculates the data size reduced by deduplication and/or compression over the original data size. A higher reduction ratio is better.
- **Throughput** measures the speed of deduplication, compression, and decompression of different systems.
- **Scalability** measures how the storage system scales with the number of models. It primarily concerns deduplication, which requires a huge volume of metadata for serving (decompression). Deduplication storage systems are known to suffer from high I/O latency due to excessive metadata overhead [12, 60, 68]. Therefore, we use metadata size as a proxy for scalability.

Setup. We conduct our experiments on an Amazon EC2 `c6a.48xlarge` instance, equipped with a 96-core AMD EPYC 7R13 processor and 384 GB of DRAM. All models and associated data are stored on an EBS SSD volume.

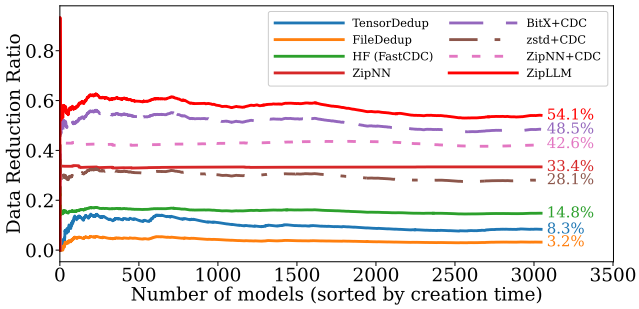


Figure 8: Data reduction ratio vs. model count.

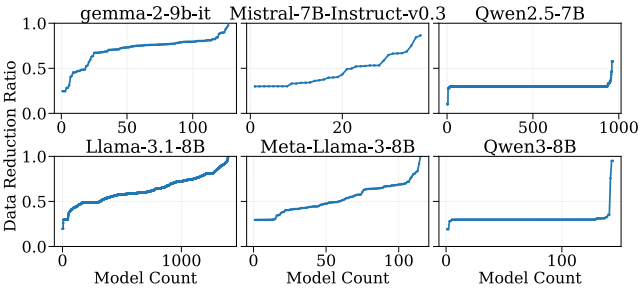


Figure 9: Data reduction ratio (DRR) distributions of six representative base models after applying BitX compression. For each base, all derived fine-tuned models are sorted by DRR in ascending order.

5.2 End-to-end Comparison

5.2.1 Data Reduction Ratio

To evaluate the overall effectiveness of ZipLLM, we run the full deduplication and compression pipeline through the entire dataset of 3,048 LLMs. To simulate the real-world scenario where users continuously upload models to a model hub like Hugging Face, we incrementally increase the number of LLMs and record the corresponding data reduction ratio. Figure 8 shows how different data reduction methods perform as the model storage scales. An ideal data reduction method would improve the ratio of redundant data with more uploaded models, leading to higher storage savings. The data reduction ratio curve reveals how quickly each method reaches its peak effectiveness and highlights the scalability and long-term benefit of ZipLLM, which continues to improve and converges later than all baselines.

Figure 8 shows that FileDedup achieves limited data reduction, eventually reducing total storage by 3.2% on the entire dataset. Upon closer examination of the redundant files, we find that most of them happen because the same models are stored multiple times by different users. Reducing deduplication granularity often improves the deduplication ratio, as it exposes fine-grained redundancy within files. ChunkDedup provides up to 4.6× (since $\frac{14.8}{3.2} \approx 4.6$) higher data reduction ratio compared to FileDedup. However, as we show next in §5.2.2, ChunkDedup incurs a significant computational and storage overhead, making it difficult to scale. ZipNN, a state-of-the-art model-aware compression algorithm, reduces model storage by 33%, much lower than ZipLLM.

Table 4: Data ingestion and retrieval throughput with 192 threads.

Method	Ingestion (MB/s)	Retrieval (MB/s)
HF (FastCDC)	2,560	9,573
ZipNN	1,424	9,663
ZipLLM	5,893	7,872

The synergy between model-aware deduplication and compression allows ZipLLM to further boost the data reduction ratio to 54.1%. This is because ZipLLM exploits LLM family information and delta-compresses fine-tuned models. ZipLLM uses *dedup-then-compress*, which outperforms *compress-then-dedup*. As shown, BitX+FastCDC, ZipNN+FastCDC, and zstd+FastCDC *(compress-then-dedup)* reach 48.5%, 42.6%, and 28.1%, respectively, confirming that compressing first hides redundancy and reduces deduplication effectiveness.

Per Family Compression Breakdown. To better understand how compression effectiveness varies across different model families, we break down the result and present the data reduction ratio distributions of six representative LLM families in Figure 9. We find that ZipLLM achieves large benefits for most families (e.g., Gemma and Llama-3.1), with median reduction above 0.4-0.7. In contrast, for the Qwen series, the results are more diverse: Qwen includes multiple base variants (e.g., math [64], coder [65], and VL [66]), and user-provided model cards are often incomplete. These factors make grouping less precise, leading to a DRR close to zstd-level compression. In the future, we plan to improve our grouping algorithm to better handle such heterogeneous families.

5.2.2 Throughput Performance

Data Ingestion Throughput. When the model storage system receives a model upload request, it must perform deduplication and compression before writing the data to the storage. Although the processing can be asynchronous, the data ingestion speed reflects the computational cost required to store models efficiently. FileDedup is the most performant solution due to its simplicity. As a result, its throughput is bottlenecked by the I/O bandwidth. HF (FastCDC) achieves 2,560 MB/s (Table 4), which is much slower. Moreover, CDC requires sequential boundary detection using a rolling hash, further limiting its scalability. ZipNN, on the other hand, is the slowest among the three, reaching only 1,424 MB/s (Table 4).

Because TensorDedup only requires calculating a hash for each tensor, it can scale linearly with the number of tensors. ZipLLM's data ingestion throughput only depends on BitX compression. We find that it can achieve a compression speed of over 5,893 MB/s.

Data Retrieval Throughput. While computation during data ingestion is performed only once, the computational cost during model downloading—including decompression—must be incurred at each model serving. Since retrieving deduplicated data incurs almost no overhead, we primarily focus on decompression. As a common baseline for decompression, the state-of-the-art generic lossless compressor zstd achieves a

Table 5: Deduplication statistics. TensorDedup strikes a balance between data reduction ratio and overheads. The projected HF metadata size is based on 17 PB of models hosted by Hugging Face in 2024 [79], while the estimated metadata size is based on the sampled 3,048 LLMs.

Deduplication Level	Unique Hashes	Avg Size (MB)	Max Size (MB)	Reduction Ratio	Throughput (MB/s)	Estimated Metadata (MB)	Projected HF Metadata (GB)
ChunkDedup (FastCDC)	520,551,953	0.087	0.25	14.8%	2560	31,772	12,505
TensorDedup (ours)	923,384	44.9	2,000	8.3%	39,690	56	22.1
LayerDedup	96,643	433.2	4,096	5.4%	38,990	5.9	2.4
FileDedup	12,465	3,824	15,316	3.2%	27,099	0.76	0.30

single-threaded decompression throughput of 1050 MB/s on our testbed, since its decoding process cannot be parallelized. In contrast, both FastCDC and ZipNN achieve over 9 GB/s (9,573 MB/s and 9,663 MB/s, respectively), while ZipLLM achieves 7,872 MB/s (Table 4). These rates are well above typical disk or network bandwidth, indicating that decompression is not the bottleneck during model retrieval.

5.3 Breakdown Analysis

In this section, we break down ZipLLM into deduplication and compression for detailed benefit analysis.

5.3.1 Deduplication

To compare different deduplication strategies, we evaluate four methods—FileDedup, LayerDedup, TensorDedup, and ChunkDedup (FastCDC)—on our 3,048-model dataset. We have discussed FileDedup in the previous section, so we focus on the others in this section.

ChunkDedup. As shown in Table 5, ChunkDedup achieves the highest data reduction ratio, removing up to 14.8% of total data across all models. However, ChunkDedup is relatively slow and produces a huge amount of metadata. For example, 520,551,953 unique chunks were produced in our 3,048-model dataset. This results in a vast corpus of metadata to be cached in memory for fast access. Hugging Face stores over 17 PB of models in 2024 [79]. Assuming each chunk requires 64 bytes of metadata [12][5], ChunkDedup requires over 12.5 TB of storage to just store the metadata. If they were stored in memory with `c6a.48xlarge` EC2 VMs (384 GB of DRAM) it would require at least 33 VMs. Production systems often replicate metadata for high availability, which would further increase the resource usage. For the same reason, IBM reports that chunk-level deduplication is impossible to deploy at scale [59]. Worse, large metadata overhead is known to degrade system performance [12, 60, 68].

Another key limitation of CDC-based ChunkDedup is its selection of chunk size. To balance between the deduplication ratio and overheads, CDC typically uses large chunk sizes (e.g., 64 KB in Hugging Face [81]). However, model tensors typically range from a few KB to hundreds of MB, meaning that a single tensor often spans multiple chunks, and chunk boundaries may not align with tensor boundaries. This misalignment not only causes boundary-shifting but also complicates the use of downstream model-structure-dependent compressors, which would require extra mechanisms and overhead to recover alignment. In contrast, knowing the data consists of LLMs, we leverage the structural information, such as tensors, to perform more effective, structure-aligned TensorDedup.

TensorDedup. Because tensors are 100-1000× larger than chunks, TensorDedup produces only 923K unique hashes across the same dataset—a three-order-of-magnitude reduction compared to ChunkDedup (Table 5). Meanwhile, it achieves a data reduction ratio of 8.3%. This leads to a dramatically smaller metadata index size—approximately 22.1 GB for all the models stored in Hugging Face. TensorDedup enables more scalable deduplication in terms of memory footprint and system manageability. By leveraging explicit tensor boundaries in formats like `safetensors`, it eliminates the need for rolling hash computations and boundary detection. Moreover, TensorDedup is highly parallelizable as each tensor can be processed independently. As a result, TensorDedup achieves 15× higher throughput compared to ChunkDedup.

LayerDedup. Besides TensorDedup, model-awareness also enables another level of deduplication—LayerDedup. An LLM typically consists of multiple layers, each of which has multiple tensors. LayerDedup performs deduplication at a higher level with coarser granularity, allowing it to generate even fewer entries with less metadata. However, only 5.4% of the data can be reduced using LayerDedup.

To validate our results, we conduct a direct comparison with the CDC results reported by the Hugging Face Xet team [80] on a public repository [8] used in the blog [80]. In Hugging Face production, the 191 GB model was reduced to 97 GB using CDC. Our FastCDC baseline and TensorDedup both report 93 GB after deduplication. First, this confirms that TensorDedup achieves a comparable data reduction ratio. Second, the production ChunkDedup result is slightly higher than our experimental results, which we conjecture is due to the production system using a larger chunk size to reduce overhead.

Visual Comparison. We randomly select a model and apply different deduplication methods. As shown in Figure 10, CDC (ChunkDedup) and TensorDedup produce very similar results. The only major difference appears in the embedding tensor. This is likely due to vocabulary expansion in fine-tuned models. Although the embedding dimension may change, most of the vocabulary stays the same. Due to its finer granularity, CDC can still match portions of the embedding bytes. In

[5] This is a reasonable assumption, as metadata typically includes chunk hashes, locations, permissions, reference counts, and timestamps.

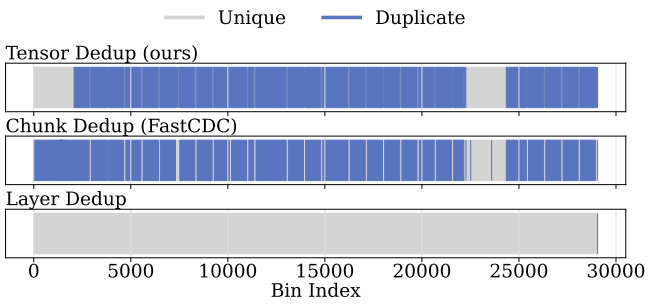


Figure 10: Visualization of deduplication results under three deduplication levels on a randomly selected LLM repository. Blue indicates duplicate content, while gray indicates unique data.

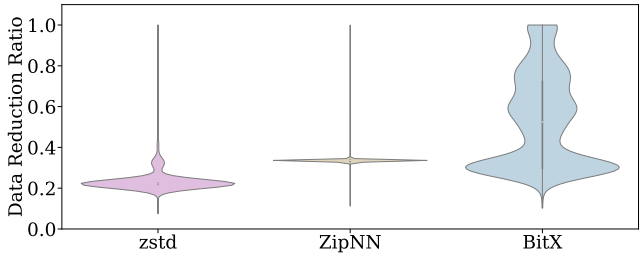


Figure 11: Distribution of data reduction ratio using different compression methods. Each violin plot illustrates the density and spread of data reduction ratios per method, overlaid with a box plot that marks the interquartile range and median.

contrast, TensorDedup treats each tensor as a whole—any small change makes the entire tensor non-deduplicable. For the same reason, LayerDedup misses most redundancy because a single modified tensor breaks the whole layer. Even so, TensorDedup covers nearly all remaining redundancy in the model. It performs as well as CDC, but with significantly less metadata and higher throughput.

5.3.2 Lossless Compression

Figure 11 shows the data reduction distribution across all 3,048 models using three lossless compression methods: BitX (ours), ZipNN, and zstd. Compared to model-oblivious zstd, model-aware compression algorithms, such as ZipNN and BitX, achieve significantly higher data reduction ratios. Between ZipNN and BitX, we observe that BitX achieves the best overall data reduction ratio, with many model sizes being reduced by over 50%. The main advantage of BitX over ZipNN is the XOR-based delta compression, which is more effective at compressing models from the same LLM family.

BitX is both more effective and more performant. Because ZipNN groups the sign, exponent, and mantissa from all floating-point numbers in the model for Huffman encoding, it cannot be effectively parallelized, leading to a lower compression throughput as shown in Table 4. Because BitX operates at the tensor level, each tensor can be processed independently and in parallel, significantly improving compression throughput and achieving over 4× the throughput of ZipNN.

These results demonstrate that BitX is both effective and practical for large-scale model storage, achieving significantly better compression while preserving losslessness and model fidelity. As previously shown in Table 4, BitX also achieves the highest compression throughput, making it well-suited for online model storage reduction at scale.

6 Discussion

Encouraging Standardized Tensor Naming and Ordering. While `safetensors` provides a safe and efficient format for model serialization, its specification allows flexibility in tensor naming and does not require preserving the original serialization order. In practice, many models use custom naming conventions or reorder tensors alphabetically by name, which can complicate BitX matching that relies on consistent tensor alignment. We advocate adopting more standardized practices—such as unified naming schemes and optionally recording tensor serialization order—would make it easier to identify corresponding tensor pairs, thereby improving compression effectiveness in systems like ZipLLM.

Online Quantization and Model Storage Co-design. We observe that many LLM repositories include multiple `GGUF` files that differ only by quantization method. These variants are often derived from the same base model. This redundancy could be avoided by storing only the base model and the quantization configuration. The backend can then perform online quantization to generate the desired quantized variant on demand, trading additional computation for greater storage savings. *We believe these findings open new research avenues in storage-efficient model quantization and encourage further exploration of new ML system design principles that co-designs quantization techniques with storage backends.* This approach is not only storage-efficient but also offers flexibility for future quantization schemes without requiring re-uploads of the same base model.

Cost Savings and Practical Impact. To understand real-world benefits, we estimate the storage cost reduction achievable by our approach. According to Hugging Face's public statistics, the total model storage footprint reached around 17 PB in 2024 [80]. If ZipLLM achieves a 50% reduction, this would save approximately 8.5 PB of capacity. Assuming standard Amazon S3 pricing [7], this translates to an annual cost saving of more than $2.2M.

7 Conclusion

This paper presents ZipLLM, a model storage reduction pipeline that unifies tensor-level deduplication and a new lossless delta compression called BitX to address the growing scale of LLM storage. Our large-scale study reveals key redundancies in LLM repositories and motivates design principles that synergize model storage deduplication with compression. ZipLLM achieves significantly higher storage savings and throughput compared to state-of-the-art approaches, without sacrificing losslessness.

Acknowledgment

We thank our shepherd, Dushyanth Narayanan, and the anonymous reviewers for their valuable feedback and comments, which improved the paper. We thank Ajit Banerjee, Di Xiao, and Yucheng Low from the Xet team of Hugging Face for insightful discussions and feedback on this work. This research was supported in part by NSF grants CNS-2322860, OAC-2411009, and OAC-2403313. We also acknowledge support from NSF CloudBank for providing AWS credits, and thank Adobe for their generous research gift.

References

[1] HDF Software Documentation. https://support.hdfgroup.org/documentation/index.html.

[2] IEEE 754-2019: IEEE Standard for Floating-Point Arithmetic. https://standards.ieee.org/ieee/754/6210/.

[3] Model Family Tree. https://huggingface.co/posts/mlabonne/611875460328127.

[4] Building and operating a pretty big storage system called S3. https://www.allthingsdistributed.com/2023/07/building-and-operating-a-pretty-big-storage-system.html, 2023.

[5] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Evgenii Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems*, 2019.

[6] Amazon Web Services. Amazon s3: A simple storage service. https://aws.amazon.com/s3/, 2006.

[7] AWS. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/, 2025.

[8] bartowski. gemma-2-9b-it-gguf. https://huggingface.co/bartowski/gemma-2-9b-it-GGUF, 2024.

[9] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[10] Martin Burtscher and Paruj Ratanaworabhan. High throughput compression of double-precision floating-point data. In *2007 Data Compression Conference (DCC'07)*, pages 293–302. IEEE, 2007.

[11] Yann Collet and Murray Kucherawy. Zstandard compression and the application/zstd media type. Technical report, 2018.

[12] Biplob Debnath, Sudipta Sengupta, and Jin Li. {ChunkStash}: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[13] Dell Technologies. Understanding data domain compression. https://www.dell.com/support/kbdoc/en-us/000003886/86266-understanding-datadomain-compression, 2023.

[14] Dell Technologies. Powerprotect dd backup appliances. https://www.dell.com/en-us/dt/data-protection/powerprotect-backup-dd-appliances/powerprotect-dd-backup-appliances.htm, 2024.

[15] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023.

[16] Peter Deutsch. Deflate compressed data format specification version 1.3. Technical report, 1996.

[17] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[18] James Diffenderfer, Alyson L Fox, Jeffrey A Hittinger, Geoffrey Sanders, and Peter G Lindstrom. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing*, 41(3):A1867–A1898, 2019.

[19] Peijie Dong, Lujun Li, Yuedong Zhong, Dayou Du, Ruibo Fan, Yuhan Chen, Zhenheng Tang, Qiang Wang, Wei Xue, Yike Guo, and Xiaowen Chu. Stbllm: Breaking the 1-bit barrier with structured binary llms. https://arxiv.org/abs/2408.01803, 2024.

[20] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[21] Hugging Face. Hugging face. https://huggingface.co/, 2023.

[22] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[23] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design

tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 331–344, USA, 2015. USENIX Association.

[24] ggml-org. Gguf: Ggml universal file format specification. https://github.com/ggml-org/ggml/blob/master/docs/gguf.md, 2024.

[25] GitHub. Git large file storage (lfs). https://github.com/git-lfs/git-lfs, 2024.

[26] Google. TensorFlow Hub: A Library for Reusable Machine Learning Modules. https://www.tensorflow.org/hub, 2018.

[27] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and Ahmad Al-Dahle et al. The llama 3 herd of models. https://arxiv.org/abs/2407.21783, 2024.

[28] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[29] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. https://arxiv.org/abs/2403.14608, 2024.

[30] Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, and Danny Harnik. Zipnn: Lossless compression for ai models. *arXiv preprint arXiv:2411.05239*, 2024.

[31] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[32] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[33] Hugging Face. Model cards - hugging face documentation. https://huggingface.co/docs/hub/en/model-cards, 2024.

[34] Hugging Face. Safetensors documentation. https://huggingface.co/docs/safetensors/en/index, 2024.

[35] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. https://arxiv.org/abs/2310.06825, 2023.

[36] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. Btrblocks: Efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[37] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment*, 15(11):3058–3070, 2022.

[38] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 179–189. IEEE, 2018.

[39] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[40] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[41] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment*, 14(11):2586–2598, 2021.

[42] James Liu, Guangxuan Xiao, Kai Li, Jason D Lee, Song Han, Tri Dao, and Tianle Cai. Bitdelta: Your fine-tune may only be worth one bit. *Advances in Neural Information Processing Systems*, 37:13579–13600, 2024.

[43] Josh MacDonald. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science . . . , 2000.

[44] Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. Context-based adaptive binary arithmetic coding in the h. 264/avc video compression standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):620–636, 2003.

[45] Meta AI. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024.

[46] Meta AI. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/, 2024.

[47] Meta AI. Llama 3.1 8b. https://huggingface.co/meta-llama/Llama-3.1-8B, 2024.

[48] Meta AI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/, 2024.

[49] Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025.

[50] Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

[51] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 1, USA, 2011. USENIX Association.

[52] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. https://arxiv.org/abs/2402.06196, 2025.

[53] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[54] NetApp. Netapp ontap 9 storage efficiency guide. Technical Report TR-3966, NetApp, 2020.

[55] NetApp. Ontap data management software. https://www.netapp.com/data-management/ontap-data-management-software/, 2024.

[56] Fan Ni and Song Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *Proceedings of the ACM symposium on cloud computing*, pages 220–232, 2019.

[57] Ivica Nikolic, Teodora Baluta, and Prateek Saxena. Model provenance testing for large language models. https://arxiv.org/abs/2502.00706, 2025.

[58] Wanyi Ning, Jingyu Wang, Qi Qi, Mengde Zhu, Haifeng Sun, Daixuan Cheng, Jianxin Liao, and Ce Zhang. Fm-delta: Lossless compression for storing massive fine-tuned foundation models. *Advances in Neural Information Processing Systems*, 37:66796–66825, 2024.

[59] Myoungwon Oh, Sungmin Lee, Samuel Just, Young Jin Yu, Duck-Ho Bae, Sage Weil, Sangyeun Cho, and Heon Y. Yeom. TiDedup: A new distributed deduplication architecture for ceph. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 117–131, Boston, MA, July 2023. USENIX Association.

[60] Yanqi Pan, Wen Xia, Erci Xu, Hao Huang, Xiangyu Zou, and Shiyi Li. Don't maintain twice, it's alright: Merged metadata management in deduplication file system with GogetaFS. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 479–495, Santa Clara, CA, February 2025. USENIX Association.

[61] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[62] William B. Pennebaker, Joan L. Mitchell, Glen G Langdon, and Ronald B Arps. An overview of the basic principles of the q-coder adaptive binary arithmetic coder. *IBM Journal of research and development*, 32(6):717–726, 1988.

[63] Colin Percival. bsdiff - binary diff/patch utility. https://www.daemonology.net/bsdiff/, 2003.

[64] Qwen. Introducing Qwen2-Math. https://qwenlm.github.io/blog/qwen2-math/, 2024.

[65] Qwen. Qwen2.5-Coder: Code More, Learn More! https://qwen.ai/blog?id=d9c66f64e7a2e156790c7991df3c803a7c3f96cd&from=research.research-list, 2024.

[66] Qwen. Qwen2.5 VL! Qwen2.5 VL! Qwen2.5 VL! https://qwen.ai/blog?id=c5e7415d9a9e89adc18c59d9e466e5a1a459b8f4&from=research.research-list, 2025.

[67] Morgane Riviere et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[68] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 24, USA, 2012. USENIX Association.

[69] Michal Stabno and Robert Wrembel. Rlh: Bitmap compression technique based on run-length and huffman encoding. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 41–48, 2007.

[70] Zhaoyuan Su, Ammar Ahmed, Zirui Wang, Ali Anwar, and Yue Cheng. Everything you always wanted to know about storage compressibility of pre-trained ml models but were afraid to ask. *Proc. VLDB Endow.*, 17(8):2036–2049, April 2024.

[71] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[72] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[73] Andrew Tridgell et al. Efficient algorithms for sorting and synchronization. 1999.

[74] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.

[75] Deepak Vohra. Practical hadoop ecosystem. *Chapter in Apache Parquet*, 177:178, 2016.

[76] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 4, USA, 2012. USENIX Association.

[77] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[78] Wikipedia contributors. Vcdiff – wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/VCDIFF, 2024.

[79] Xet Team. Git LFS Usage across the Hub. https://huggingface.co/spaces/xet-team/lfs-analysis, 2024.

[80] XetHub. From Files to Chunks: Improving HF Storage Efficiency. https://xethub.com/blog/from-files-to-chunks-improving-hf-storage-efficiency, 2024.

[81] XetHub. From chunks to blocks: Accelerating uploads and downloads on the hub. https://huggingface.co/blog/from-chunks-to-blocks, 2025.

[82] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. {FastCDC}: A fast and efficient {Content-Defined} chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

[83] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[84] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[85] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[86] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 110–127, 2025.

[87] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.

[88] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

[89] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[90] Sally Zhu, Ahmed Ahmed, Rohith Kuditipudi, and Percy Liang. Independence tests for language models. https://arxiv.org/abs/2502.12292, 2025.

[91] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

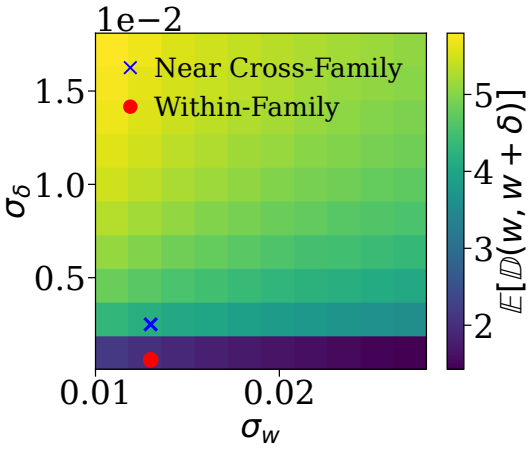


Figure 12: Expected bit distance heatmap.

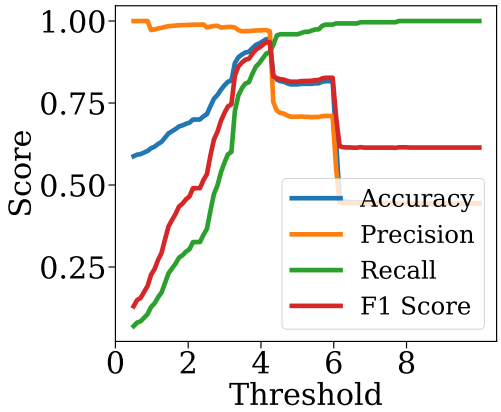


Figure 13: Impact of selected threshold on various metrics.

A Appendix

A.1 Sensitivity of Clustering Threshold

As discussed in §3.4.3, we use a clustering threshold on the bit distance to determine whether a model pair belongs to the same family.

To select a robust and interpretable threshold, we refer to the empirical parameter distribution of popular model families (e.g., `Llama-3.1`, `Mistral`, `Qwen2.5`, etc.), where the standard deviations of base weights and fine-tuning deltas typically fall within $\sigma_w \in [0.01, 0.05]$ and $\sigma_\Delta \in [0.00, 0.02]$ (see Figure 12). In this heatmap, darker colors indicate higher expected bit distance values. Under this distribution, the expected bit distance lies within $[1.5, 6]$ based on Monte Carlo sampling. In contrast, cross-family pairs typically exceed 6 due to larger weight deltas, as shown in Figure 3.

A notable exception is model pairs that are closely related but cross-family (e.g., `Llama-3` vs. `Llama-3.1`), which show a lower-than-expected bit distance around 4 (red dot in Figure 12). This near-cross-family case reveals a potential risk: although a higher threshold (e.g., 6) would be suggested based on the general distribution, it could lead to false negatives by misclassifying closely related but cross-family models. To address this, we narrow the threshold down to 4, which effectively mitigates such risks. As shown in Figure 13, the threshold of 4 achieves a high accuracy of **93.5%**, while maintaining a good balance across precision (by avoiding misclassifying cross-family models as within-family), recall (since thresholds below 4 would miss many true within-family pairs), and thus results in a strong F1 score.