FlatStor: An Efficient Embedded-Index Based Columnar Data Layout for Multimodal Data Workloads

Chi Zhang
Shanghai Jiao Tong University
C_Zhang1996@sjtu.edu.cn

Shihao Zhang
Shanghai Jiao Tong University
zsh_henry200135@sjtu.edu.cn

Yunfei Gu
Shanghai Jiao Tong University
gu.yunfei@sjtu.edu.cn

Chentao Wu*
Shanghai Jiao Tong University
wuct@sjtu.edu.cn

Jie Li
Shanghai Jiao Tong University
lijiecs@sjtu.edu.cn

Qin Zhang
Huawei Cloud
kevin.zhangqin@huawei.com

Xusheng Chen
Huawei Cloud
chenxusheng6@huawei.com

Jie Meng
Huawei Cloud
mengjie09@huawei.com

ABSTRACT

Modern data lakes have become essential for storing, managing, and analyzing massive amounts of heterogeneous data. As production data increasingly exhibits multimodal storage characteristics and multi-purpose access patterns, efficient management of such complexities becomes critical. However, current hybrid storage system-based data lakes face persistent challenges, including synchronization overhead, data correlation disruption, and escalating storage costs due to the involvement of multiple underlying storage systems. While columnar storage, central to data lakes, addresses hybrid-system inefficiencies, it struggles with the complexities of multimodal data storage and multi-purpose access.

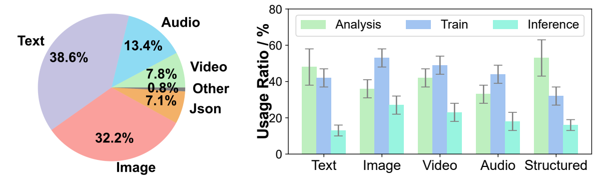
To tackle these challenges, we analyze access patterns across various scenarios and assess the issues in storing multimodal data. Based on these insights, we propose FlatStor, a FlatBuffers-based columnar Storage format with embedded indexing. It supports point access through indexing and handles multimodal data by vertically partitioning and treating each modality as a byte stream for storage. It also applies FSST compression, reducing storage overhead significantly. Benchmark evaluations reveal that FlatStor reduces the access latency by 99.6% and the storage overhead by 91.3% compared to Parquet in inference workloads. Furthermore, FlatStor outperforms LanceV2 with a 41.3% latency improvement, maintaining minimal additional overhead.

PVLDB Reference Format:
Chi Zhang, Shihao Zhang, Yunfei Gu, Chentao Wu, Jie Li, Qin Zhang, Xusheng Chen, and Jie Meng. FlatStor: An Efficient Embedded-Index Based Columnar Data Layout for Multimodal Data Workloads. PVLDB, 19(1): 1-14, 2025.
doi:10.14778/3772181.3772182

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-Storage-Lab/FlatStor-Columnar-Layout.

*Corresponding Author

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772182



(a) Intrinsic multimodality (b) Cross-application utilization patterns

Figure 1: Modern production data landscape

1 INTRODUCTION

In the big data era, enterprises increasingly struggle with storing and analyzing massive, heterogeneous data. Data lake technology has emerged as an effective solution [43, 53], providing distributed storage for raw data while integrating modern query engines to enable efficient data management and analysis [24–28, 30, 38]. At the core of this technology lies columnar storage, with mainstream formats like Parquet [7] and ORC [6, 44] becoming de facto industry standards. These formats are widely supported by distributed storage systems (e.g., HDFS [62], S3 [2] and processing frameworks (e.g., Spark [8], Flink [5]), as well as major cloud data warehouses including Redshift [33, 42, 58] and Snowflake [23, 39].

Recent studies underscore inherent limitations of conventional columnar storage in meeting emerging application demands [45, 51, 64]. Figure 1 highlights two key characteristics of modern production data at petabyte scale: the *multimodal nature* at storage and the *multi-purpose access* at the application layer, where identical data serve multiple departments for diverse purposes. Multimodality introduces distinctive storage challenges, including massive aggregate volumes, numerous small items, and predominantly item-level access. Existing solutions mitigate these complexities by integrating specialized storage systems for different modalities alongside columnar architectures. For example, H2O [31] deploys mixed layouts and dynamically adjusts formats at runtime. Nevertheless, this approach has notable drawbacks. First, it incurs considerable synchronization overhead due to frequent format transformations. Second, it disrupts data correlations by dispersing data across systems, requiring intricate metadata management to maintain holistic relationships. The fundamental challenge is achieving efficient multimodal storage while preserving accessibility and integrity.

The data utilization patterns in Figure 1b highlight another challenge: in production, identical data are often reused across multiple organizational units for three main functions—analytical processing by BI teams, model training by ML engineers, and real-time inference in operational systems (e.g., RAG scenarios using internal data). This tripartite usage produces varying access patterns, from batch-oriented analytical queries to latency-sensitive point-access inference, all on shared data. Current implementations rely on department-specific copies, causing substantial storage overhead—each unit maintains independent replicas, often with extra copies for high availability—leading to 3–5× higher raw storage needs. Interconnected workflows exacerbate latency due to continuous synchronization across isolated storage, reducing throughput. These inefficiencies highlight the need for storage architectures that support diverse access patterns while ensuring data integrity, optimizing resources, and reducing operational complexity.

Given the proven success of columnar storage in data lake implementations for analytical and model training workloads [32, 55, 66], it is natural to explore whether its architecture can be extended to natively support multi-purpose multimodal data storage. A unified columnar approach simplifies system design and offers the potential to: (1) eliminate unnecessary data synchronization, (2) preserve critical data correlations, and (3) reduce storage costs by minimizing redundant replication. The key challenge is enabling columnar systems to efficiently support diverse, dynamic access patterns for multimodal data while retaining benefits such as low storage overhead and high query performance. This challenge can be distilled into two core problems: 1) identifying and managing access patterns across applications and 2) analyzing multimodal data layouts to uncover inefficiencies in columnar usage.

To address limitations in existing multimodal storage, we propose FlatStor, a storage format based on FlatBuffers that extends columnar storage with embedded indexing and multimodal-aware partitioning. Unlike prior work treating multimodal data as isolated vectors or using coarse partitioning, FlatStor applies fine-grained vertical partitioning, dividing data into modality-specific columns and treating each data item as a byte stream, enabling unified, efficient storage of heterogeneous data. It also embeds an index for point access, reducing inference latency without sacrificing throughput. Additionally, FlatStor uses FSST for deduplication, balancing storage efficiency with fast decoding and access.

Our contributions are fourfold: 1. summarizing common multimodal data access patterns across applications; 2. revealing limitations of existing hybrid methods in handling such access; 3. proposing FlatStor, a columnar layout with embedded indexing addressing point access and multimodal storage challenges; 4. implementing and evaluating FlatStor on multiple benchmarks. Experiments show that, in inference scenarios, FlatStor achieves significantly lower access latency than Parquet and LanceV2 [18], with minimal overhead. These results validate FlatStor as a practical, effective solution for diverse multimodal data access, bridging the gap between traditional columnar storage and modern workload demands.

The remainder of this paper is organized as follows: Section 2 presents background and motivation; Sections 3 and 4 describe the design and implementation of FlatStor; Section 5 presents experimental results and analysis; Section 6 discusses related work; and Section 7 concludes.

2 INSIGHTS & MOTIVATIONS

This section analyzes application access patterns, evaluates existing hybrid schemes and their limitations for multi-purpose multimodal data, examines issues in columnar storage with multi-granularity access, and highlights challenges in multimodal data.

2.1 Access Patterns Across Diverse Applications

By analyzing the I/O processes of various application categories, we summarize their key characteristics as follows:

Data Analytics. In big data analytics, I/O is dominated by batch access, where systems sequentially read and process large data volumes for operations like aggregation and filtering, involving continuous high-volume reads with minimal random access.

Model Training. During training, data are accessed via batch-based requests. Each batch retrieves a contiguous subset of data, which is shuffled locally for randomness before GPU training. The I/O pattern follows a periodic batch access pattern, as batches are requested at regular intervals.

Model Inference. Inference tasks like RAG [50] often start by retrieving a few relevant records—facts, memories, or documents—via vector search. These *top-k* results point to scattered data entries, producing random point-access patterns. Such fine-grained, unpredictable access is essential for context-aware, accurate responses. Inference also spans multiple levels, from point access for fact lookup to batch access for complex reasoning [67], showing that multiple I/O granularities coexist in these scenarios.

In summary, despite varied applications, I/O falls into two types: batch access and point access. This classification guides storage system design to balance batch tasks (like analytics) and real-time inference (like RAG [50]).

2.2 Limitations of Existing Approaches

We divide current storage schemes into two types: **Discrete Storage (DS)** and **Aggregated Storage (AS)**. DS involves minimal preprocessing before direct storage, as in file, object, and key-value stores like MinIO [19], Redis [21], and RocksDB [22]. AS aggregates and processes data before storage, including row- and column-based formats such as Parquet [7], ORC [6], Arrow IPC [4], SnowFlake [23], RedShift [33], DuckDB [11], PostgreSQL [20], and Aurora [63]. Based on access pattern analysis in Section 2.1, we evaluate AS and DS using the LAION-Aesthetics dataset [60, 61], following Section 5. Raw data is stored in MinIO (DS) and aggregated into Parquet (AS), accessed via Arrow [3], enabling evaluation of hybrid system performance across multiple schemes.

Overall Performance. Although training workloads are generally constrained by throughput, latency remains a critical factor during cache warm-up phases and in environments with resource contention, both of which affect training stability. As shown in Figure 2, during model training (batch query), AS reduces latency by 46% compared to DS. Conversely, during inference (point query), DS reduces latency by 97% relative to AS. This significant contrast indicates that neither AS nor DS alone can fully satisfy the latency and throughput requirements across different usage scenarios, underscoring the necessity for a unified and flexible storage solution.

Table 1: Comparison between AS and DS

	Discrete Storage	Aggregated Storage
Storage Overhead	High	Low
Metadata Overhead	High	Low
Management Difficulty	Medium	Medium
System Portability	Medium	High

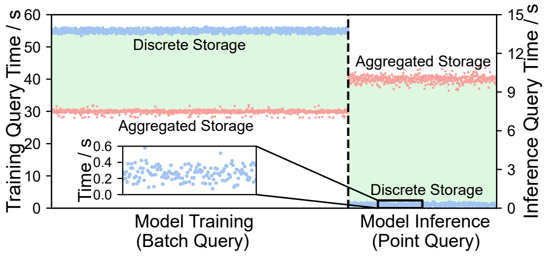


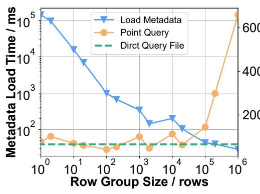
Figure 2: I/O performance of AS vs DS across applications

Data Synchronization Overhead. Data conversion experiments show that converting the dataset from AS to DS takes **4,800 CPU hours**, while the reverse takes **7,400 CPU hours**. Since LAION-Aesthetics is a small subset, overhead grows with scale. This reveals a key flaw in hybrid systems—their inability to adapt to changing needs without heavy resources. The problem worsens when data serves both analysis and inference, as repeated conversions create bottlenecks and complexity. These results highlight the need for a unified storage system that supports diverse access patterns without repeated conversions.

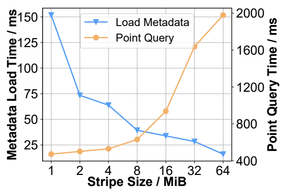
Disruption of Data Correlation. While hybrid storage solutions can meet diverse access needs, they often cause *Loss of Data Correlation*. When data is fragmented across formats, cross-field or cross-row analysis becomes more complex. Frequent migrations also increase overhead and risk data integrity, reducing query efficiency. In contrast, a unified format can preserve data relationships, enabling efficient multi-dimensional analysis and complex queries, offering a more robust solution.

Substantial Storage Overhead. Hybrid multi-storage schemes often require maintaining separate high-availability replicas for different formats to ensure functionality during failures. This redundancy multiplies storage overhead, as the same data must be stored in multiple formats. For example, supporting both analytical and inferential workloads may demand full data copies in both KV and columnar storage, driving up a typical 3-5× increase in storage overhead. A unified storage format eliminates redundant replication, cutting costs and simplifying system design.

Table 1 compares additional metrics, showing that DS incurs higher storage and metadata overhead due to independent data item management, which complicates compression. In contrast, AS offers greater portability with consistent data definitions and standardized interfaces like *Schema* and *SQL*, ensuring adaptability across infrastructures. Based on these findings, AS is the preferred solution. It can be divided into row- and columnar formats. Given the superior efficiency of columnar storage in filtering and tasks essential for analysis and model training via selective access and better compression, it is chosen as the final format.



(a) Parquet



(b) ORC

Figure 3: Point query performance between Parquet and ORC

2.3 Constraints of Columnar Storage

2.3.1 ***Poor point access performance.*** In Section 2.2, we analyze limitations in current hybrid approaches and explain the rationale for choosing columnar storage as the base solution. This section also investigates key factors affecting point access performance. Figure 3 presents performance of mainstream columnar formats under point access and their metadata loading overhead, using the same setup as Figure 2. In Figure 3a, the green dashed line indicates baseline performance when accessing raw data directly from remote object storage such as MinIO. Results show that increasing row group (or ORC stripe) size lowers metadata overhead but significantly worsens access performance. Due to small test case scale, ORC stripe count varies less than Parquet row groups, leading to smaller metadata size and better access performance for ORC. Because a column chunk within a row group is the minimal accessible unit, smaller row groups reduce chunk size and I/O amplification, narrowing the gap with raw data. Based on the connection among chunk size, I/O amplification, and observed results, we conclude that I/O amplification is the primary factor behind degraded performance. This conclusion broadly holds for columnar formats like Parquet, ORC, Arrow IPC, Snowflake, Redshift, and DuckDB, all using the row group concept.

2.3.2 ***Layout-related motivations.*** An intuitive way to improve point query performance is by reducing column chunk size, but this is constrained by the row group configuration, which applies uniformly across the table. Once data is written, changing this setting typically requires reloading and rewriting the table, causing high overhead and leaving row group size mostly fixed. Many analytical applications favor large row groups for sequential loading—Parquet recommends 512 MiB to 1 GiB, while ORC defaults to 64 MiB to match HDFS [62] block sizes. Existing systems also rely on column chunks for metadata like Bloom filters and dictionaries. Smaller chunks increase metadata overhead, explaining the sharp rise in metadata loading seen in Figures 3a and 3b as chunk size shrinks. Additionally, current columnar layouts inherit legacy structures from relational databases, using row groups originally to partition tables for easier management. While this enables partitioning without detailed schema knowledge, it mismatches columnar usage—users seldom access entire tables or row groups but rather specific columns or chunks. This misalignment, combined with tight coupling of row groups to I/O, leads to excessive I/O amplification in point access. These problems highlight the need to decouple data management from access granularity to better support efficient point queries and flexible access, such as inference.

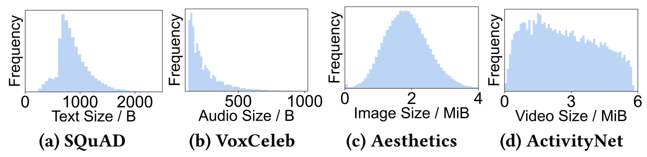


Figure 4: Multimodal data size distributions across datasets

2.4 Inside Multimodal Dataset

2.4.1 ***Data characteristics.*** Our work focuses on efficient storage and access of multimodal data items such as single images, audio clips, or video segments, while tasks like video segmentation and alignment are performed upstream and lie outside the scope. To analyze data characteristics, we select four representative datasets: SQuAD [59] (text QA), VoxCeleb [56] (speech), LAION Aesthetics (images), and ActivityNet [36] (video). Figure 4 shows item size distributions, with even video rarely exceeding 6 MiB. Total storage ranges from hundreds of GiB to several TiB, implying vast numbers of items. Due to semantic alignment challenges [41], multimodal data is usually processed via single-modality training and late-stage fusion, so most datasets contain no more than three modalities (text, image, audio). In summary, modern datasets exhibit three traits: *small item size, large volume*, and *limited modalities*, posing two key challenges: **storage efficiency** and **multimodal support**.

2.4.2 ***Data storage efficiency.*** Data storage efficiency addresses the challenges of large volumes of small data items in current datasets, aiming to minimize storage costs. These costs consist of metadata and data. In Section 2.2, columnar storage is selected as the foundational solution through comparative analysis, with Table 1 demonstrating its lower metadata costs. Consequently, the primary remaining challenge lies in reducing data storage costs.

In columnar storage, data in the same column often share similar types and characteristics, making compression a common and effective way to reduce storage costs. Compression methods generally fall into two types: *dictionary-based* and *statistic-based*. Dictionary-based compression replaces repeated symbols with references from a dictionary. The algorithm builds a dictionary of unique symbols and assigns each an index. Repeated symbols are then replaced with indices, reducing overall size. Decompression restores the original data using the dictionary. Statistic-based compression uses data properties such as frequency and symbol probabilities to assign shorter codes to frequent symbols, minimizing encoded length. An encoding table is created for compression, and decompression uses it to restore the original data.

Both compression algorithms typically process large datasets globally for optimal compression ratios, requiring global decompression to restore the original data. Figure 5 shows the size of Parquet files generated from part of the LAION-Aesthetics dataset using different compression algorithms, along with point access performance for a specified column. While compression reduces data size, it can degrade access performance due to global decompression. Recent studies [49, 64] suggest that as storage costs decrease, compression methods should prioritize decompression efficiency to improve access performance. Emerging algorithms like FSST [35] support random access and partial decompression, but rely on explicit boundary information during data retrieval. Poor metadata design can lead to the loss of boundary information, forcing random queries to load entire compressed blocks, even if decompression itself is not the bottleneck.

2.4.3 ***Multimodal support.*** Current methods for storing multimodal datasets typically save data URLs as strings while storing raw data in separate systems, as exemplified by LAION, JourneyDB [57], and HuggingFace [15]. In Section 2.2, we analyze the shortcomings of hybrid storage systems and advocate for a single columnar storage solution. However, directly storing multimodal data in columnar formats leads to the *row group explosion* problem, where atomic multimodal data (e.g., videos, images) cannot be column-split or selectively accessed. Fine-grained atomic access necessitates small row groups, resulting in excessive metadata overhead.

2.4.4 ***Data-related motivations.*** Based on the above analysis, regarding data storage efficiency, directly applying compression algorithms in columnar storage may introduce extra access constraints beyond the row group issue, further degrading performance in inference scenarios. Thus, metadata design should support random accessible compression methods and store explicit boundary information to enable advanced compression. For multimodal data storage, this concern aligns with the fine-grained access performance issues analyzed in Section 2.3, further emphasizing the core role of row groups in current designs and the need to decouple data management from access in columnar storage. To address this, we introduce a vertical partitioning method that treats each modality type as an independent column. Each column can adopt its own management strategy and data access granularity, free from unified row group constraints. This ensures that even with reduced granularity, the row group explosion problem is avoided. More details are provided in Section 3.2.

Table 2: Challenges and Corresponding Solutions in FlatStor

Challenges	Solutions
Balancing fine-grained access with metadata overhead	Vertical partitioning enables independent access granularity per column and overcomes row group limitations.
Relatively large data storage overhead	Random-accessible compression preserving granularity and access storage efficiency.
Row group explosion due to atomic access requirements for modality data	Multi-level fine-grained metadata design reduces redundant metadata overhead.

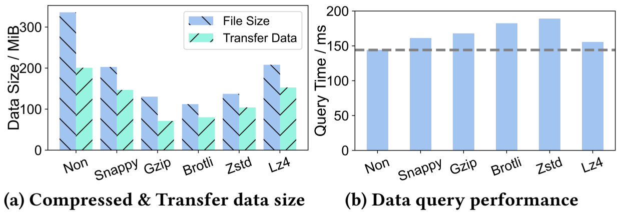


Figure 5: Parquet query performance under different compression algorithms

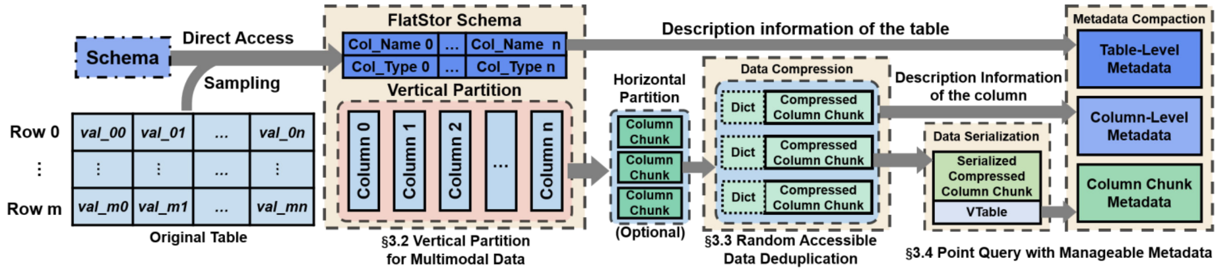


Figure 6: Workflow Generation in FlatStor

3 FLATSTOR

In this section, we first provide a brief overview of the generation process of FlatStor, a FlatBuffers-based columnar Storage format with embedded indexing for efficient multimodal data access. This is followed by a detailed explanation of each step in the workflow, covering all stages from Section 3.2 to Section 3.4.

3.1 Overview

FlatStor introduces a novel columnar layout that enables efficient point queries and multimodal data storage, while preserving the benefits of traditional columnar formats and minimizing overhead. This is achieved by addressing three key challenges: 1) supporting multimodal data, 2) reducing storage overhead, and 3) enabling efficient point access without significantly increasing metadata overhead, thus avoiding the problem illustrated in Figure 3. To this end, FlatStor adopts three core designs: Prioritized Vertical Partitioning, Random Accessible Deduplication, and Serialization with Metadata Compaction, as shown in Figure 6. A brief overview is provided below:

(1) *Vertical Partition for Multimodal Data:* To support multimodal storage, FlatStor adopts column-based vertical partitioning over row-based horizontal partitioning. This enables independent management of each column by data type, free from unified table constraints, and allows better optimization for heterogeneous data.
(2) *Random Accessible Data Deduplication:* To reduce storage overhead, FlatStor uses FSST [35] as its core compression method for efficient and fast data encoding. FSST compresses data more effectively than LZ4 [68, 69] and supports random access, ensuring that compression does not hinder fine-grained access granularity or performance.
(3) *Point Query with Manageable Metadata:* To enable efficient point access with controlled metadata overhead, FlatStor serializes column chunks. During serialization, data offsets are recorded via an embedded index to facilitate point access. Metadata is organized in three tiers (table, column, and column-chunk levels) and compacted by scope, reducing unnecessary overhead through reuse.

By integrating these methods, FlatStor achieves configurable access granularity, manageable metadata overhead, and support for multimodal data storage. Figure 6 illustrates the workflow, and the subsequent sections provide detailed designs for each step.

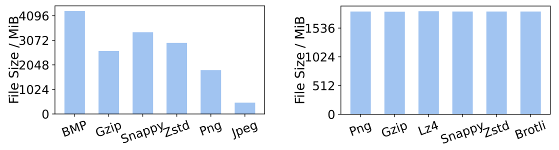
3.2 Vertical Partition for Multimodal Data

In current columnar storage, tables are typically divided horizontally into smaller subtables called *row groups*, which are further split by column to form *column chunks*. This partitioning reduces data management costs and improves concurrency and reliability. However, it forces the entire table to share the same horizontal partitioning, requiring column chunks to align with row groups. Since each column stores different data types, configurations should vary. Uniform horizontal grouping makes it difficult to ensure stable performance across columns or meet specific needs of diverse data types. For instance, differences in data scale and encoding between *string* and *image* columns prevent optimizing I/O performance for both, favoring only one grouping size. Furthermore, partitioning is typically done once in both directions, so the number of rows in a chunk is set by horizontal partitioning. Row groups often contain multiple rows for management efficiency, limiting fine-grained access to column chunks and creating a tradeoff between data access performance and metadata overhead, as shown in Figure 3.

To address limitations of globally consistent row groups in the current design, FlatStor redefines table partitioning by reversing the order of horizontal and vertical partitioning. This adjustment introduces several key benefits: ***Simplified Data Hierarchy***: Prioritizing vertical partitioning conceptualizes the table as a collection of column units, aligning naturally with columnar storage principles. *Customized Horizontal Partitioning*: Each column independently determines its horizontal partitioning size based on its data characteristics, such as modality. Combined with modality-specific configurations (e.g., encoding, indexing), this ensures optimal performance per column. *Granular Metadata Design*: Managing data at both table and column levels enables more detailed metadata, supporting efficient and flexible data management. Further details on metadata design are discussed in Section 3.4.

Although columnar organization is preferred by default in FlatStor for easier storage, input tables from other systems are often row-organized, requiring conversion. Depending on table type, we apply different strategies to process *records* and *fields* from the original table to generate the FlatStor schema:

- *Table with schema.* This type of table is already properly formatted, allowing direct extraction of descriptive information from the original schema. Typical examples include DataFrames or relational database tables. We can directly retrieve column names, data types, and other field details and convert them into column descriptions to generate a new schema in FlatStor.



(a) IMG with different compression methods

(b) PNG with multiple compression techniques

Figure 7: Comparing Parquet's image storage efficiency across compression algorithms

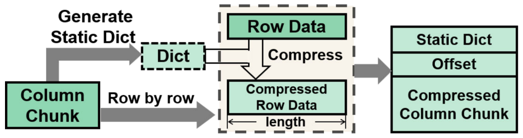


Figure 8: The compression process of FSST

- *Table without schema.* Another scenario involves receiving tables without column descriptions—for example, fragments from CSV files or table-structured data parsed from network byte streams. In such cases, we cannot directly obtain column information for a FlatStor schema. Instead, we assign placeholder names like "Column N" by record order. To infer data types, we perform random sampling on each column, initially treating all columns as *string*, then adjust types based on sampled data features—for instance, assigning *double* if decimal points appear. This enables construction of a basic FlatStor schema.

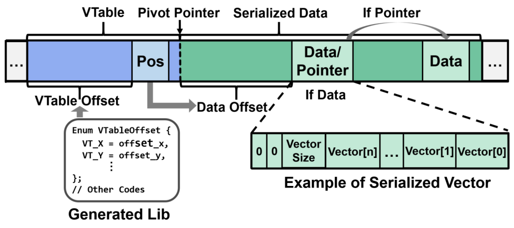
Once the schema for FlatStor is obtained using the above methods, we create vectors for each column to temporarily store its data. By iterating through the table row by row, we parse each record and append values to the corresponding vectors. This yields the schema and associated column vectors in FlatStor, completing vertical partitioning. Each column is now stored independently, allowing tailored management and configuration. Although multimodal data is stored as byte streams, original encoding formats include *magic numbers* as identifiers, distinguishing *string* from modal data and resolving modality at the storage level. The upper layer can define access interfaces for different modalities, but such designs lie beyond the scope of this paper.

3.3 Random Accessible Data Deduplication

After resolving multimodal data storage challenges, the next issue is storage overhead. As noted in Section 2.4.1, most datasets include no more than three modalities—typically text, images, and audio [41]. To address storage overhead for modality-specific complex data, we use image data as an example and apply the method in Section 3.2, storing it as *string* data in Parquet files for analysis. Figure 7 shows our results. For raw image data (BMP in Figure 7a), mainstream compression methods in columnar storage fall far short of image-specific techniques. Some methods (e.g., Lz4, Brotli) even fail on images, showing that modality-specific data requires specialized encoding or deduplication to reduce storage overhead. Moreover, Figure 7b shows that after converting images to PNG,

Table 3: Parquet file sizes for the same dataset under varying row group configurations

Row group Size	1	10	100	1,000	10,000
File Size	5.4 GiB	941.3 MiB	404.5 MiB	346.7 MiB	333.0 MiB



```
Enum VTableOffset {
  VT_X = offset_x,
  VT_Y = offset_y,
        ⋮
};
// Other Codes
```

Figure 9: Serialized data layout with FlatBuffers-enabled random access

no compression method achieves further reduction, indicating that once modal data undergoes format-specific deduplication, general-purpose compression offers no additional gains.

For simple modality data like *string*, FlatStor primarily uses FSST (Fast Static Symbol Table) for compression. FSST is a lightweight, efficient algorithm for *string* data, employing a *static symbol table* to encode frequent substrings, achieving high compression ratios with fast compression and decompression. Unlike traditional dictionary methods, FSST allows access to individual items within a compressed block without traversing the entire block, making it ideal for low-latency scenarios. Figure 8 shows the FSST compression process. During compression, FSST records each item's length, converted to a global offset within the data chunk. Because the dictionary is static, decompression uses these offsets to locate and decompress the needed item directly, enabling random access.

It is important to emphasize that the data deduplication method in FlatStor is applied independently to each modality-specific column, which are managed separately. This independent deduplication does not cause loss of correlation between modalities, because modalities remain aligned at the record level through unified schema definitions and row index consistency. The schema explicitly records correspondences between modalities, similar to how Parquet handles the LAION dataset. The deduplication method in FlatStor combines the two approaches mentioned above. As described in Section 3.2, after vertical partitioning, independently managed columns and their corresponding vectors are obtained. *Magic numbers* determine whether data in each vector is modality-specific complex data and apply the appropriate deduplication method. For modality-specific complex data, deduplication treats each data item independently within the same column, without comparing different items. After deduplication, the number of items remains unchanged, and each item remains independently accessible. For simple modality data like *string*, FSST ensures random access after compression, relying on offsets, which may fail with poor layout, and row group limitations can hinder access. Therefore, the data format is adjusted to accommodate offsets and avoid row group restrictions, as discussed in Section 3.4. For other simple types, such as *int* and *double*, compression methods supporting random access are similar to FSST, so specific methods are not detailed here.

3.4 Point Query with Manageable Metadata

The final major challenge for FlatStor is supporting random point access in inference scenarios while minimizing metadata overhead. Table 3 shows Parquet file sizes for the same data sample under different row group sizes. As noted in Section 2.3, the row group is the core design element in most current columnar storage systems [1, 4, 6, 7, 11, 39], with the column chunk inside a row group as the smallest unit for management and access. This design causes metadata overhead to rise sharply as access granularity decreases—smaller chunks mean more chunks to manage for the same data volume. The core issue stems from tight coupling between data management and access mechanisms in existing systems. Therefore, addressing this requires decoupling data management from data access in columnar storage.

Inspired by KV storage and FSST, FlatStor uses a serialization-based embedded indexing method to decouple data management from access granularity. Traditional KV systems support basic set and get operations, limiting direct access to data inside values. By serializing values in a specific format, internal data becomes accessible, treating a column chunk as a value and each row as internal data within it. FSST's offset recording similarly enables direct access via offsets. FlatStor embeds offset indices for each row within a column chunk using serialization, enabling random access while retaining block-based management. The VTable records offsets of each item within the serialized data, enabling random access. Since offsets are stored in a pre-generated static lib file, parsing avoids reading irrelevant VTable data, reducing overhead.

After decoupling data management from access in FlatStor and stabilizing management overhead regardless of access granularity, we further reduce metadata overhead through consolidation. Besides the growing number of column chunks, uncontrolled metadata records cause significant redundancy. For example, data columns often use dictionary encoding to improve storage, but metadata suffers from two redundant designs: 1) at the row group level, where each row group maintains multiple dictionaries; and 2) at the data page level, where each page uses an independent dictionary. These ignore data similarity within columns, causing duplicate dictionaries and extra I/O overhead, especially at the page level. Thus, we propose that metadata components should be set at appropriate scales to improve efficiency. We classify metadata types as follows:

- **Schema**: A schema describes the overall structure of a two-dimensional dataset, such as a table. It defines a sequence of fields with names, data types, and optional metadata. For multimodal data, the schema preserves alignment semantics by assigning each modality to a dedicated field and enforcing consistent row-wise ordering. Cross-modal relationships—such as temporal or semantic alignment—can be encoded through schema metadata to support coordinated interpretation and access.
- **Dictionary**: The dictionary is used for dictionary encoding, a technique that represents data values by referencing unique entries in a dictionary. This method is particularly effective when data contains many duplicate values. During encoding, values are represented by an array of non-negative integers that index the corresponding entries in the dictionary.

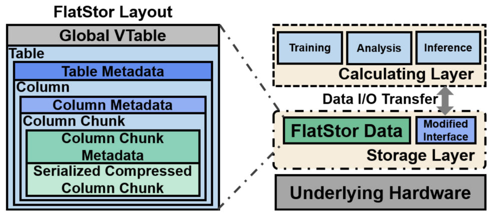


Figure 10: Integrating FlatStor with storage system

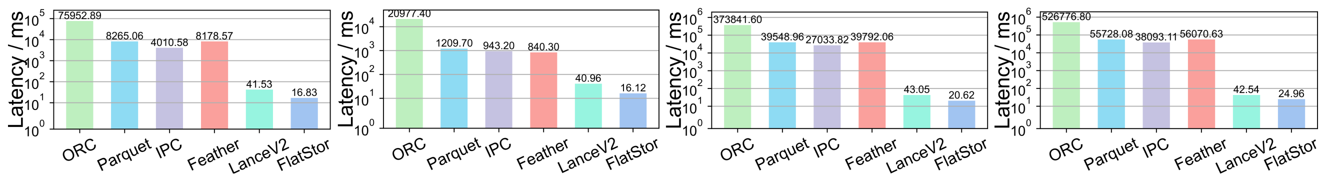
- **Statistics**: Statistical information includes data such as maximum and minimum values, as well as the number of rows in a column chunk. During a range query, a Bloom filter can efficiently sift through column chunks that meet the criteria based on the statistical information, thereby enhancing data retrieval efficiency.
- **Others**: Other information, such as data addresses, data versions, and other customized configurations, should also be included in the metadata. This type of information pertains to user-defined configurations.

FlatStor organizes metadata into three tiers: *Table*, *Column*, and *Column Chunk*. The **Schema** belongs in table-tier shared metadata. **Dictionaries** may reside in table-tier metadata if column data types are similar; otherwise, they are in column or column chunk metadata. **Statistics** belong to column or column chunk metadata. **Others** are placed according to their association with the data. As noted in Section 2.3, most columnar storage systems require loading and rewriting the entire dataset for updates, leading to poor write performance. Consequently, updates and deletions (which may trigger metadata changes) are infrequent, and although FlatStor's metadata compaction increases modification cost, the impact remains acceptable.

After completing data deduplication in Section 3.3, we observe that for a column chunk vector, regardless of the deduplication method, we obtain a deduplicated vector matching the original length. For example, during FSST compression, its output can be adjusted to form a vector of compressed row data based on recorded offsets. Deduplicated vectors from different columns are integrated using the serialization method described here, ensuring each column has the same VTable-Serialized Data structure. In serialization, original FSST offsets can be replaced by a VTable with the same function, enabling synergy with FSST format design. Deduplicated vectors from modality-specific complex data require no special handling. We then consolidate metadata across columns; for instance, encoding dictionaries from various chunks are traversed to extract common parts into column-level metadata, leaving only unique parts in chunk metadata, which are reconstructed during decoding.

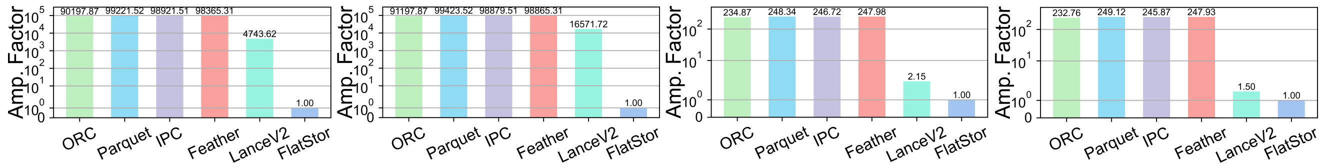
4 PUT IT ALL TOGETHER

After processing a table through the generation workflow described in Section 3, we have obtained metadata structured into three tiers, along with several serialized and compressed column chunks. Since serialization is a widely adopted technique for both metadata management and data format definition [9, 14], we consistently apply unified serialization to efficiently handle and store both the three-tier metadata and the corresponding data chunks across the system.



(a) SQuAD (Text) (b) VoxCeleb (Audio) (c) LAION Aesthetics (Image) (d) ActivityNet (Video)

Figure 11: Query latency of different formats under inference workload scenario



(a) SQuAD (Text) (b) VoxCeleb (Audio) (c) LAION Aesthetics (Image) (d) ActivityNet (Video)

Figure 12: Query amplification of different formats under inference workload scenario

Figure 10 (left half) shows the final physical layout of FlatStor after serializing three-tier metadata alongside data, following a bottom-up hierarchy. At the lowest level, vectors correspond to *Column Chunks*, with metadata stored alongside (*Column Chunk tier*). Multiple column chunks form column data, with metadata completing the *Column tier*. Columns combine into table data, stored adjacent to table metadata (*Table tier*). The *Global VTable* records offsets of each table part, enabling efficient random access. The right half illustrates FlatStor integration with storage systems (e.g., data lakes), replacing existing data formats and extending access interfaces for point access, supporting inference while remaining compatible with traditional workflows.

5 EVALUATION

In this section, we evaluate our design using various metrics, including data access performance across different workload scenarios, ablation studies, and overhead analysis.

Methodology. Our baselines include Parquet [7] and ORC [6], two widely used columnar formats, along with IPC and Feather [4]. IPC is the columnar format in Arrow [3], while Feather is its serialization format using FlatBuffers. We also compare with LanceV2, the format used by LanceDB [17, 18], a container-based database optimized for multimodal data. As LanceV2 is still under development, we implement its core container design based on available documentation for testing. To ensure consistency across formats, Arrow is used as the default data access tool during evaluation.

Metrics. We compare performance across multiple metrics: I/O performance in inference, analytics, and model training scenarios; data skipping access efficiency; prioritized vertical partitioning impact; deduplication effectiveness; serialization impact; Arrow integration adaptability; sensitivity analysis; and overhead analysis of I/O requests, metadata storage and access latency.

Platform. We deploy a MinIO [19] storage cluster as remote storage and use a compute node for access experiments. The cluster has four machines, each with dual Intel® Xeon® E5-2620 CPUs, 64 GiB memory, and 23.8 TB HDD, running Ubuntu 20.04.6 LTS with GNU/Linux 5.15.0-105-generic x86_64, connected via 100 Mbps LAN. The compute node has dual HiSilicon KunPeng 920 CPUs, 128 GiB memory, 4 TiB HDD, and 4 TiB NVMe SSD, running Ubuntu 20.04.6 LTS with GNU/Linux 5.4.0-193-generic aarch64. Network bandwidth between the compute node and cluster is 1 Gbps. For local experiments, tests run directly on the compute node with local storage. System buffers and network caches are reset before each round to eliminate caching effects.

Dataset. We evaluate using the datasets in Figure 4, preprocessed via byte-stream conversion from Section 3.2 to meet experimental requirements. SQuAD [59] is a large-scale reading comprehension dataset with crowdworker-generated questions and text-extracted answers from Wikipedia. VoxCeleb [56] contains celebrity speech recordings from YouTube. LAION Aesthetics [60], a subset of LAION, includes images rated for aesthetic quality. ActivityNet [36] is a large-scale video understanding dataset with annotated videos of diverse human activities.

Configuration. The default configurations for different file formats, based on the datasets above, are as follows: Parquet, IPC, and Feather (via Arrow) use a row group size of 100,000 for SQuAD and VoxCeleb, or 250 for LAION Aesthetics and ActivityNet to control file size, while ORC's stripe size matches Parquet's row group count. Other settings, such as encoding and pre-cache strategies, follow Arrow's defaults. FlatStor uses the same configurations as Parquet. LanceV2's container size is set to 4 MiB. File sizes for all formats are kept below 1 GiB, and compression algorithms are avoided unless explicitly stated to prevent access performance degradation.

5.1 Inference Workload Testing

Many inference pipelines, especially retrieval-augmented ones (e.g., RAG [50]), use top-k vector search to find relevant items from a preprocessed corpus, with retrieved IDs resolved into full data entries, causing random point access. We use FAISS for vector retrieval, converting top-k results into data accesses via a reverse

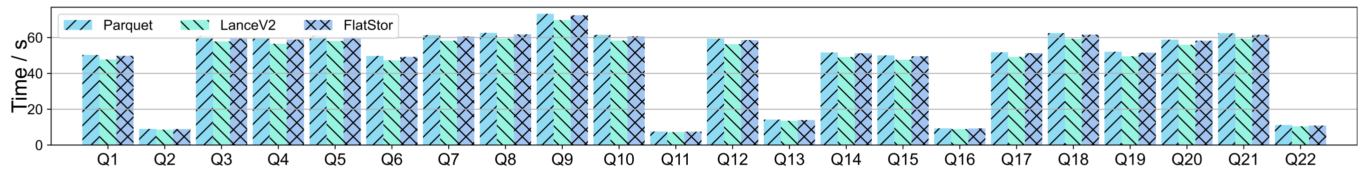


Figure 13: TPC-H benchmark results of FlatStor, Parquet and LanceV2

Table 4: MLPerf Storage benchmark results of FlatStor, Parquet and LanceV2

Workload	**Dataset Size / GiB**	**Samples #. / s**		
		Parquet	**LanceV2**	**FlatStor**
Unet3D	270	16.728	17.102	16.875
CosmoFlow	211	798.598	809.136	803.613

index to raw data in remote storage. Vector retrieval is common across methods and excluded from our results. Figures 11 and 12 show average access latency and read amplification for various datasets and formats. Parquet, ORC, IPC, and Feather exhibit poor access due to high read amplification and excessive I/O. LanceV2 accesses fixed-size containers, yielding stable latency but worse amplification for small objects (SQuAD, VoxCeleb). FlatStor supports random access via VTable, maintaining low read amplification and reducing data transfers, lowering latency by approximately 99.6% versus traditional methods and 41.3–61.6% versus LanceV2, demonstrating superior efficiency.

5.2 TPC-H Benchmark

In this subsection, we use DuckDB to run the TPC-H benchmark. Table data with scale factor sf=100 is generated and converted into various file formats. For each query, Arrow reads files directly or converts data into Arrow Tables. Relevant tables are loaded into a new DuckDB instance to avoid interference from existing tables, and SQL is executed. Results cover the entire process, from reading data in different layouts to completing TPC-H tasks. Figure 13 shows results for Parquet, LanceV2, and FlatStor. FlatStor performs similarly to Parquet. While LanceV2's container size is optimized for storage I/O, frequent offset retrieval reduces access performance. In contrast, FlatStor keeps detailed index info for column chunks and columns in the VTable, enabling efficient reading and consistently strong analytical performance.

5.3 MLPerf Benchmark

This subsection uses MLPerf Storage [34] to benchmark training workload performance. Due to limited format support in MLPerf, we enhanced compatibility of its underlying *dlio_benchmark* [10] for testing. Experiments run on a single node with four A100 accelerators. Table 4 shows the size of pre-generated training sample files, ensuring datasets prevent DRAM caching effects. MLPerf Storage measures performance in samples per second (samples/s). Results in Table 4 show FlatStor maintains a consistent sample loading rate compared to Parquet and LanceV2, preserving columnar storage benefits for training workloads.

Table 5: Access performance of different storage methods combined with serialization.

Storage Methods	**Point Access / ms**		**Batch Access / ms**	
	Caption	**Image**	**Caption (1,000 Rows)**	**Image (100 Rows)**
Row-based	5.13	36.47	95.34	3731.82
Horizontal (row group 100)	5.45	36.98	83.23	**3429.81**
Horizontal (row group 1,000)	5.22	37.92	**69.47**	3594.76
FlatStor	5.34	37.34	**72.76**	**3448.43**

5.4 Impact of Prioritized Vertical Partitioning

This subsection evaluates the improved flexibility in data management and access granularity enabled by prioritized vertical partitioning. Deduplication and serialization methods from Section 3 are also applied to row-based and traditional prioritized horizontal partitioning systems, for which demo versions are created. Using 10,000 caption-image pairs from LAION-Aesthetics, we assess point and batch access for specific columns. In prioritized horizontal partitioning, row groups are uniformly set to 1,000 or 100, while FlatStor sets caption column chunks to 1,000 and image chunks to 100. Table 5 shows performance results. Although serialization enables point access in all approaches, row-based storage performs worst for batch access due to row-by-row parsing. Traditional prioritized horizontal partitioning, with a single row group configuration, only supports efficient batch access for one column at a time. In contrast, FlatStor's prioritized vertical partitioning allows independent configuration of batch access granularity per column, enabling efficient batch access across multiple columns simultaneously.

5.5 Data Skipping Access

This subsection evaluates data skipping performance across file formats. Data skipping, a key advantage of columnar storage, enables quick column retrieval via column names or Bloom filters on column chunks. The caption column from LAION Aesthetics is shuffled into a 10-column, 1,000,000-row table with unique column names, pre-converted into multiple uncompressed formats and stored in a remote cluster. Row group size is 100,000, with LanceV2 container size 4 MiB. Figures 14a and 14b show total latency for accessing column chunks and columns. Feather does not support column chunk access. LanceV2 returns data at container granularity during chunk skipping, with increased filter overhead. FlatStor achieves performance comparable to Parquet and other formats, preserving the benefits of columnar storage.

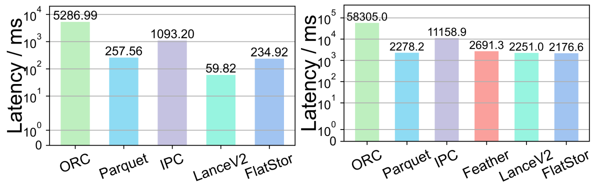


Figure 14: Data skipping access performance of different file formats

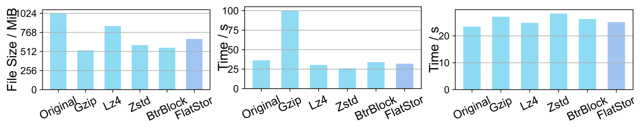


Figure 15: Compression efficiency compared to Parquet with different compression algorithms.

5.6 Data Deduplication Efficiency

This subsection evaluates the effectiveness of the data deduplication and compression scheme in FlatStor. The encoding and compression methods for modality-specific complex data follow standard practices and apply to formats like converting PNG to JPEG or using H.265 for videos. Thus, the compression process in FlatStor is consistent with conventional approaches. The focus is on how the FSST algorithm reduces storage overhead. Figure 15 compares file size and read/write performance using FSST in FlatStor with Parquet using no compression, gzip, lz4, zstd, and BtrBlock, based on the test table from Section 5.5. FlatStor reduces file size by 33.5%, improves compression by about 17%, and maintains access speed close to LZ4. BtrBlock achieves higher compression but with longer decoding time, making FlatStor more suitable for balanced workloads.

5.7 Impact of Serialization

In this subsection, we select the raw string data from the *caption* column of LAION Aesthetics. By randomly sampling sentences and truncating them, we generate data items of varying sizes (16B, 64B, 256B, 1KiB, 4KiB) and create five single-column tables of length 100,000 based on these items to evaluate the impact of item size on serialization.Additionally, we generate single-column tables of lengths 1, 10, 100, 1,000, 10,000, and 100,000 based on 4KiB-sized items to assess the impact of item count on serialization. Figure 16 presents our test results. The overhead caused by the FlatBuffers serialization in FlatStor depends solely on the number of items. Using 32-bit storage, maximum additional overhead per table is approximately $row_{num} \times column_{num} \times 4$ bytes. Since columns in FlatStor are managed independently, it is possible to configure whether each column requires embedded indexing to support point access. In scenarios where indexing is not needed (e.g., treating column chunks or column as a single data block), the additional overhead will be lower than the estimated value.

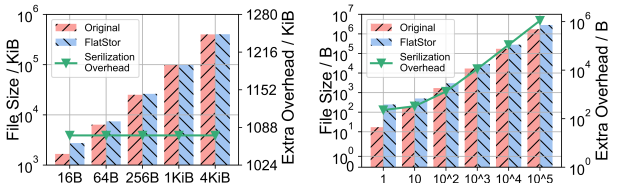


Figure 16: Analysis of factors affecting serialization overhead

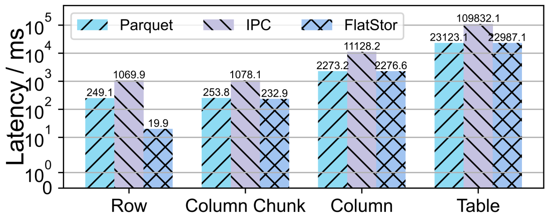


Figure 17: Integration capability testing of FlatStor and Arrow.

5.8 Adaptability to Arrow Integration

This subsection evaluates FlatStor integrated with Arrow using the testing table from Section 5.5. *Apache Arrow* provides a standardized columnar memory layout, enabling high-speed data transfer and processing across computing frameworks without memory copying, enhancing analysis efficiency and interoperability. Figure 17 demonstrates the performance of FlatStor with Arrow integration across access granularities. It achieves an 8.3% improvement in column chunk access compared to Parquet, and maintains similar column-level performance. It also excels in point access (row access), outperforming Parquet by about 9.65x. These findings align with prior experiments, confirming that integrating FlatStor into existing database or analytical systems boosts overall capabilities. This integration allows enterprises to support fine-grained access in new scenarios while preserving their current toolchains.

5.9 Sensitivity Analysis

FlatStor stores modality-specific complex data as byte streams (strings), making access to different data types equivalent to accessing string data (search is not discussed here). This subsection analyzes sensitivity by focusing on the impact of column count, rows per column, and size distribution on data I/O. Three sets of local test cases use the method from Section 5.7: 1) tables with 1,000 rows and columns numbering 10, 20, 50, 100, 200, 500, and 1,000, with fixed item size 1 KiB; 2) tables with 10 columns and rows numbering 10, 100, 1,000, 10,000, and 100,000, fixed item size 1 KiB; and 3) three single-column tables, each with 10,000 rows, data sizes from 16 B to 4 KiB, following uniform, Zipfian, and Gaussian distributions. Native Parquet lacks single item access, so to enable it, row group size is set to 1, greatly increasing metadata overhead—used as control. Except for the data distribution test (where Parquet row group size is 10,000), reads are at the column level and writes at the table level without compression. Results average multiple rounds.

Table 6: I/O performance under varying conditions (FlatStor demonstrates stable performance)

Column Num	Writing Table / ms			Reading Column/ ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
10	591.62	587.34	639.31	560.35	527.85	544.94
20	1743.89	1711.75	1799.34	567.03	551.15	553.66
50	5735.29	5713.34	5857.65	552.23	545.44	549.36
100	9628.83	9632.56	9925.71	579.76	558.22	564.88
200	22799.64	22725.45	24214.45	556.45	548.23	547.88
500	62641.98	62623.45	66116.87	598.75	583.11	585.04
1000	112822.51	112691.12	125382.45	557.34	549.15	553.86

Row Num	Writing Table / ms			Reading Column / ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
10	5.34	4.31	5.24	5.46	5.38	5.51
100	47.51	43.34	46.31	36.47	32.83	34.12
1000	591.34	587.45	609.31	529.85	512.08	514.39
10000	6948.31	6934.34	7025.34	5483.73	5290.13	5351.35
100000	79834.97	78681.45	80251.98	52082.09	51342.45	51752.93

Data Distribution	Writing Table / ms			Reading Item / ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
Uniform	192.12	189.34	194.45	25357.892	46.653	19.652
Zipfan	112.45	101.89	108.54	23262.012	42.893	17.532
Gaussian	234.12	213.76	223.13	24693.456	44.423	18.982

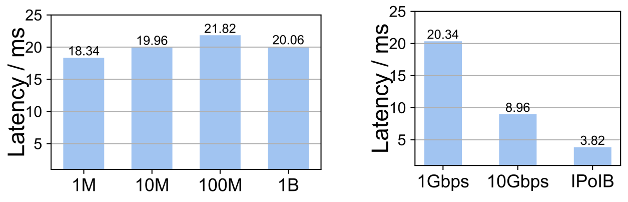
Table 6 presents I/O performance results for Parquet, LanceV2, and FlatStor under various column settings, row counts, and size distributions. As the number of columns increases, FlatStor experiences growing overhead from merging metadata across columns, leading to greater write latency than the other two methods. This higher write latency limits the applicability of FlatStor in scenarios with large numbers of columns. Under the same column configuration, all three methods show similar write latency, as shown in the middle and bottom tables. For read latency, LanceV2 benefits from its configuration of data access units based on containers, achieving better I/O performance than Parquet and FlatStor across different table shapes, while FlatStor consistently performs better than Parquet. When handling varying data distributions, Parquet must use a row group size of 1 to enable point access, which results in high metadata read costs and poor read performance. LanceV2, with containers holding multiple items as its minimum access unit, performs worse in data access compared to FlatStor, which supports finer-grained access to individual items.

5.10 Scalability Analysis

This subsection evaluates the scalability of FlatStor through two parts. First, we analyze how dataset size impacts the data point access performance of FlatStor, using subsets of the LAION-5B dataset at different scales. Then, we also evaluate point access performance under three network environments—1 Gbps, 10 Gbps, and high-performance IPoIB—while keeping the rest of the configurations same as Section 5.1. The results are shown in Figure 18. Figure 18a shows that access performance remains stable as dataset size increases. This is due to a simple preprocessing strategy, where large-scale data is pre-split into files of around 1–2 GiB before being stored in FlatStor, helping to maintain consistent access performance. Figure 18b presents the performance under the three network conditions. As network speed increases, data processing latency becomes a more visible bottleneck, suggesting further optimization opportunities in metadata handling and format parsing.

Table 7: Number of I/O operations of different formats for data at different levels

I/O Operations Number	Access Metadata			Access Data
	Table	Column	Column Chunk	
Parquet	1	1	/	>1
LanceV2	1	2	/	>1
FlatStor	1	2	3	>1



(a) Increasing Dataset Size (b) Scaling Network Bandwidth

Figure 18: Scalability under Increasing Dataset Size and Network Bandwidth

5.11 Overhead Analysis

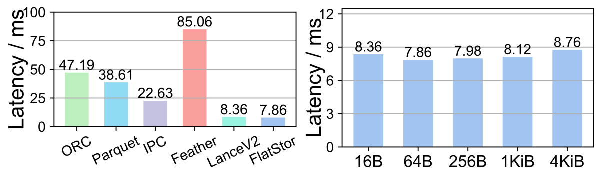
In this subsection, we evaluate the overhead of FlatStor from the perspectives of the number of I/O operations, metadata storage, and metadata access.

5.11.1 ***Number of I/O operations.*** Table 7 presents the number of I/O operations needed by Parquet, LanceV2, and FlatStor when accessing various file parts. The first two formats lack dedicated column chunk metadata. In Parquet, metadata is stored centrally, so only one access retrieves table or column metadata. LanceV2 manages data via containers; accessing table metadata requires one container, while column metadata involves first obtaining and parsing the table metadata container. In FlatStor, accessing metadata at different levels requires sequential retrieval and parsing of the Global Table, Table Metadata, and Column Metadata, involving 1, 2, and 3 I/O operations for table, column, and column chunk, respectively. Accessing actual data requires multiple I/Os proportional to data scale. This reflects how metadata design impacts overall I/O efficiency and access latency.

5.11.2 ***Total storage overhead.*** Based on previous analysis, storage efficiency of columnar formats varies across configurations for the same dataset. Several configurations are tested using the table from Section 5.9. A row group size of 100,000 is chosen for balanced data access, while a size of 1 assesses point query capabilities in formats like Parquet. Table 8 shows file sizes under these row group sizes. Results show a clear trend: as access granularity decreases, file sizes increase. FlatStor minimizes storage overhead for point queries by encoding row offsets in metadata. Metadata overhead, shown in Figure 16, grows linearly with entries and becomes negligible for larger datasets. Under point access in inference scenarios, FlatStor reduces total storage overhead by 91.6% versus Parquet with row group size 1, demonstrating higher efficiency. Comparing FlatStor, IPC, and Feather (an IPC-based format serialized by FlatBuffers) shows that even with identical serialization techniques, different workflows lead to significantly different storage outcomes.

Table 8: File sizes of the same *Table* stored in different formats different group size

Row Group Size	Parquet	IPC	Feather	FlatStor
1	5562.0MiB	1860.3MiB	2195.3MiB	538.2MiB
10	941.3MiB	529.8MiB	481.2MiB	487.7MiB
100	404.5MiB	404.1MiB	275.1MiB	482.7MiB
1000	346.7MiB	392.4MiB	239.7MiB	482.6MiB
10000	341.2MiB	391.2MiB	237.2MiB	482.3MiB
100000	339.1MiB	391.1MiB	236.6MiB	482.1MiB
Minimum Access Granularity	**Column Chunk**	**Column Chunk**	**Column**	**Row**



(a) Metadata latency of formats **(b) Metadata latency with sizes**

Figure 19: Metadata access latency of varying formats

5.11.3 ***Metadata access latency.*** In Table 8, when the row group size is set to 100,000, the maximum storage overhead of FlatStor increases by approximately 42.2% compared to mainstream solutions. Since data compression is not used during writing, this overhead mainly comes from metadata storage and management. Metadata access overhead is evaluated by measuring the time to create a data handler object, representing complete metadata access latency. Figure 19a shows FlatStor reduces metadata access latency by 90.8% and 66.1% compared to Feather and IPC, respectively. FlatStor first initiates a small I/O request to obtain the footer offset (VTable), then accesses the needed metadata level. The access time in the figure includes the total time for these two I/O operations, representing a typical metadata access process. Thus, although FlatStor involves extra I/O operations, overall metadata access performance is not significantly affected, as shown in Figure 19b.

6 RELATED WORK

Columnar Layout. Currently, research on columnar storage focusing on AI scenarios is limited. Over the past decade, columnar formats such as the Capacitor [16] format developed by Google have been proposed and used in systems such as BigQuery [13] and Napa [29]. This format, based on technologies from Dremel [54] and Abadi [25], optimizes storage layout according to workload behavior. In 2019, YouTube developed the Artus format [37] for the Procella database management system, supporting adaptive encoding without block compression and providing O(1) addressing time for nested schemas. The DWRF format [12], a variant of ORC developed by Meta, offers improved support for reading and encrypting nested data. Recent studies have highlighted new data characteristics and I/O patterns in AI scenarios [52, 64]. However, current storage format designs primarily target analytical workloads and do not effectively address inference requirements. Hybrid systems like H2O [31] have explored the combination of multiple storage layouts to adapt to varied workloads, yet such approaches mainly target structured data and do not directly address challenges of heterogeneous multimodal data.

Data Caching. Caching mitigates read amplification by replacing amplified portions with data likely to be accessed soon, thereby improving overall read efficiency. However, during model training, data is loaded randomly to reduce model bias, which conflicts with caching's reliance on data locality. Similarly, in inference, the highly discrete and random nature of requests renders caching largely ineffective. Although Shade [48] is specifically designed as a cache for training workloads, its importance sampling method is impractical [46, 47], since determining data importance requires prior training, creating a paradox. NoPFS [40] optimizes training by sharing the random shuffle seed with cache nodes, generating the same sequence and enabling pre-caching to improve I/O performance. However, it still does not fully address read amplification during retrieval, as fetching blocks continues to load unnecessary data, limiting overall efficiency gains.

Data Compression. Another approach to improving I/O performance is data compression, which reduces read amplification by lowering I/O volume and minimizing transfer overhead. FSST [35] enables random access to compressed data using a static dictionary-based table, eliminating full block decompression but struggling with efficient random access in distributed storage due to lost boundary information and inconsistent block alignment. Zhang et al. [65] record index information within compressed data to enable GPU random access and processing of compressed data, but this approach is highly specific and incompatible with columnar storage systems. Zeng et al. [64] argue that query bottlenecks have shifted from storage to computation, suggesting future formats should avoid block compression or heavyweight encodings unless justified, to optimize both storage efficiency and computational performance.

7 CONCLUSION

This paper proposes a columnar format supporting multimodal data and random point access, while maintaining efficient analytical and training workloads. To our knowledge, no existing solution addresses such diverse demands within a unified columnar framework. We analyze traditional columnar storage limitations and propose key designs: vertical partitioning, random-accessible deduplication, and metadata compression. These improve access efficiency and reduce storage overhead. Compared to mainstream formats, our solution cuts latency by 99.6% and total storage overhead by 91.3% for random point access, while maintaining stable performance on traditional workloads. Notably, FlatStor's batch access performance is comparable to existing formats, reflecting a deliberate trade-off favoring low-latency point access over batch optimization. This enables FlatStor to serve large-scale enterprises with multimodal data and diverse usage, though its advantages may be limited in workloads dominated by frequent writes or full scans.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (Grant No. 2023YFB4502900) and by Huawei Cloud. The authors would like to express their sincere gratitude to Dr. Jiong Lou from Shanghai Jiao Tong University for insightful discussions and valuable feedback.

REFERENCES

[1] 2025. Amazon Redshift. https://aws.amazon.com/redshift/ Accessed: Feb 9, 2025.
[2] 2025. Amazon S3. https://aws.amazon.com/s3/ Accessed: Feb 9, 2025.
[3] 2025. Apache Arrow. https://arrow.apache.org/ Accessed: Feb 9, 2025.
[4] 2025. Apache Arrow Columar. https://arrow.apache.org/docs/format/Columnar.html Accessed: Feb 9, 2025.
[5] 2025. Apache Flink. https://flink.apache.org/ Accessed: Feb 9, 2025.
[6] 2025. Apache ORC. https://orc.apache.org/ Accessed: Feb 9, 2025.
[7] 2025. Apache Parquet. https://parquet.apache.org/ Accessed: Feb 9, 2025.
[8] 2025. Apache Spark. https://spark.apache.org/ Accessed: Feb 9, 2025.
[9] 2025. Apache Thrift. https://thrift.apache.org/ Accessed: Feb 9, 2025.
[10] 2025. DLIO Benchmark. https://github.com/argonne-lcf/dlio_benchmark Accessed: Feb 9, 2025.
[11] 2025. DuckDB. https://duckdb.org/ Accessed: Feb 9, 2025.
[12] 2025. The DWRF Format. https://github.com/facebookarchive/hive-dwrf Accessed: Feb 9, 2025.
[13] 2025. Google BigQuery. https://cloud.google.com/bigquery Accessed: Feb 9, 2025.
[14] 2025. Google Protobuf. https://github.com/protocolbuffers/protobuf Accessed: Feb 9, 2025.
[15] 2025. Hugging Face. https://huggingface.co/ Accessed: Feb 9, 2025.
[16] 2025. Inside Capacitor, BigQuery's next-generation columnar storage format. https://cloud.google.com/blog/products/bigquery/inside-capacitorbigquerys-next-generation-columnar-storage-format Accessed: Feb 9, 2025.
[17] 2025. LanceDB. https://lancedb.com/ Accessed: Feb 9, 2025.
[18] 2025. LanceV2 Format. https://blog.lancedb.com/lance-v2/ Accessed: Feb 9, 2025.
[19] 2025. MinIO. https://min.io/ Accessed: Feb 9, 2025.
[20] 2025. PostgreSQL. https://www.postgresql.org/ Accessed: Feb 9, 2025.
[21] 2025. Redis. https://redis.io/ Accessed: Feb 9, 2025.
[22] 2025. RocksDB. https://rocksdb.org/ Accessed: Feb 9, 2025.
[23] 2025. Snowflake. https://www.snowflake.com/en/ Accessed: Feb 9, 2025.
[24] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024
[25] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.
[26] Daniel J Abadi et al. 2007. Column Stores for Wide and Sparse Data.. In *CIDR*, Vol. 2007. 292–297.
[27] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.
[28] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.
[29] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang Chen, Ming Dai, et al. 2021. Napa: Powering scalable data warehousing with robust query performance at Google. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2986–2997.
[30] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 169–180. http://www.vldb.org/conf/2001/P169.pdf
[31] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1103–1114. https://doi.org/10.1145/2588555.2610502
[32] Michael R Anderson and Michael Cafarella. 2016. Input selection for fast feature engineering. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 577–588.
[33] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.
[34] Oana Balmau. 2022. Characterizing I/O in machine learning with mlperf storage. *ACM SIGMOD Record* 51, 3 (2022), 47–48.
[35] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.
[36] Fabian Caba Heilbron, Victor Escorcia, Bernard Ghanem, and Juan Carlos Niebles. 2015. Activitynet: A large-scale video benchmark for human activity understanding. In *Proceedings of the ieee conference on computer vision and pattern recognition*. 961–970.
[37] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. 2019. Procella: Unifying serving and analytical data at YouTube. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2022–2034.
[38] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, Shamkant B. Navathe (Ed.). ACM Press, 268–279. https://doi.org/10.1145/318898.318923
[39] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[40] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 92, 15 pages. https://doi.org/10.1145/3458817.3476181
[41] Chenzhuang Du, Jiaye Teng, Tingle Li, Yichen Liu, Tianyuan Yuan, Yue Wang, Yang Yuan, and Hang Zhao. 2023. On uni-modal feature learning in supervised multi-modal learning. In *International Conference on Machine Learning*. PMLR, 8632–8656.
[42] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.
[43] Ahmed A. Harby and Farhana Zulkernine. 2022. From Data Warehouse to Lakehouse: A Comparative Review. In *2022 IEEE International Conference on Big Data (Big Data)*. 389–395. https://doi.org/10.1109/BigData55660.2022.10020719
[44] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1199–1208.
[45] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.
[46] Tyler B Johnson and Carlos Guestrin. 2018. Training deep models faster with robust, approximate importance sampling. *Advances in Neural Information Processing Systems* 31 (2018).
[47] Angelos Katharopoulos and François Fleuret. 2018. Not all samples are created equal: Deep learning with importance sampling. In *International conference on machine learning*. PMLR, 2525–2534.
[48] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 135–152.
[49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2, Article 118 (June 2023), 26 pages. https://doi.org/10.1145/3589263
[50] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401
[51] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.
[52] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.
[53] Dipankar Mazumdar, Jason Hughes, and JB Onofre. 2023. The Data Lakehouse: Data Warehousing and More. *arXiv preprint arXiv:2310.08697* (2023).
[54] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568
[55] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127* (2021).
[56] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. 2017. Voxceleb: a large-scale speaker identification dataset. *arXiv preprint arXiv:1706.08612* (2017).
[57] Junting Pan, Keqiang Sun, Yuying Ge, Hao Li, Haodong Duan, Xiaoshi Wu, Renrui Zhang, Aojun Zhou, Zipeng Qin, Yi Wang, Jifeng Dai, Yu Qiao, and Hongsheng Li. 2023. JourneyDB: A Benchmark for Generative Image Understanding. arXiv:2307.00716 [cs.CV]
[58] Ippokratis Pandis. 2021. The evolution of Amazon redshift. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3162–3174.

[59] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[60] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, et al. 2022. Laion-5b: An open large-scale dataset for training next generation image-text models. *Advances in Neural Information Processing Systems* 35 (2022), 25278–25294.

[61] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114* (2021).

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[63] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[64] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.

[65] Feng Zhang, Yihua Hu, Haipeng Ding, Zhiming Yao, Zhewei Wei, Xiao Zhang, and Xiaoyong Du. 2022. Optimizing random access to hierarchically-compressed data on GPU. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[66] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*. 1042–1057.

[67] Siyun Zhao, Yuqing Yang, Zilong Wang, Zhiyuan He, Luna K. Qiu, and Lili Qiu. 2024. Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely. arXiv:2409.14924 [cs.CL] https://arxiv.org/abs/2409.14924

[68] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[69] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934