Magnus: A Holistic Approach to Data Management for Large-Scale Machine Learning Workloads

Jun Song[†] ByteDance Inc.
Jingyi Ding[†‖] Zhejiang University
Irshad Kandy ByteDance Inc.
Yanghao Lin ByteDance Inc.
Zhongjia Wei ByteDance Inc.
Zilong Zhou ByteDance Inc.
Zhiwei Peng ByteDance Inc.
Jixi Shan ByteDance Inc.
Hongyue Mao ByteDance Inc.
Xiuqi Huang[‡‖] Zhejiang University
Xun Song ByteDance Inc.
Cheng Chen ByteDance Inc.
Yanjia Li ByteDance Inc.
Tianhao Yang ByteDance Inc.
Wei Jia ByteDance Inc.
Xiaohong Dong ByteDance Inc.
Kang Lei ByteDance Inc.
Rui Shi ByteDance Inc.
Pengwei Zhao ByteDance Inc.
Wei Chen[‖] Zhejiang University

ABSTRACT

Machine learning (ML) has become a cornerstone of key applications at ByteDance. As model complexity and data volumes surge, data management for large-scale ML workloads faces substantial challenges, particularly with recent advances in large recommendation models (LRMs) and large multimodal models (LMMs). Traditional approaches exhibit limitations in storage efficiency, metadata scalability, update mechanisms, and integration with ML frameworks. To address these challenges, we propose Magnus, a holistic data management system built upon Apache Iceberg. Magnus integrates innovative optimizations across resource-efficient storage formats optimized for large wide tables and multimodal data, built-in support for vector and inverted indexes to accelerate data retrieval, scalable metadata planning with Git-like branching and tagging capabilities, and high-performance update/upsert based on lightweight merge-on-read (MOR) strategies. Additionally, Magnus provides native support and specialized enhancement for LRM and LMM training workloads. Experimental results demonstrate significant performance gains in real-world ML scenarios. Magnus has been deployed at ByteDance for over five years, enabling robust and efficient data infrastructure for large-scale ML workloads.

PVLDB Reference Format:
Jun Song, Jingyi Ding, Irshad Kandy, Yanghao Lin, Zhongjia Wei, Zilong Zhou, Zhiwei Peng, Jixi Shan, Hongyue Mao, Xiuqi Huang, Xun Song, Cheng Chen, Yanjia Li, Tianhao Yang, Wei Jia, Xiaohong Dong, Kang Lei, Rui Shi, Pengwei Zhao, and Wei Chen. Magnus: A Holistic Approach to Data Management for Large-Scale Machine Learning Workloads. PVLDB, 18(12): 4964 - 4977, 2025.
doi:10.14778/3750601.3750620

[†]Co-first authors: wuyixin.yx@bytedance.com, dingjy@zju.edu.cn. [‡]Corresponding author: huangxiuqi@zju.edu.cn. [‖]State Key Lab of CAD&CG, Zhejiang University.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750620

1 INTRODUCTION

Machine learning (ML) has been widely adopted at ByteDance, supporting key applications such as search, recommendation, and advertising [27, 33, 42]. As these applications evolve to meet the deepening of precise and personalized demands, ML workloads have grown rapidly in both scale and complexity [29, 49, 50]. ByteDance has actively explored large models to address these evolving needs. Consequently, modern ML workloads at ByteDance encompass not only classical ML models but also large recommendation models (LRMs) [13] that capture intricate user behavior sequences and large multimodal models (LMMs) [17, 22, 47] that learn from diverse data modalities, including text, images, audio, and video.

The exponential growth of model scale and the expanding range of business applications have placed significant demands on data infrastructure. ML workloads require efficient management of massive training data. Currently, ByteDance's offline training data has reached the EB level, with a daily growth of PB. This massive amount of data is not only fundamental to model training but also plays a crucial role in feature engineering and model optimizations. However, the simultaneous surge in data volume and model complexity presents significant challenges for efficient data storage and accelerating data access. A holistic data management approach is essential to support large-scale ML workloads.

Limitations. Traditional data management approaches often store ML training data directly in distributed file systems [4] or object storage [1], but these solutions face performance bottlenecks at scale. For instance, data is stored as individual files without unified table-level organization, and metadata operations become time-consuming with the proliferation of files. Moreover, the lack of update and upsert operations limits consistent data management and complicates iterative model development. To address some of these problems, open-source data lake table formats, such as Apache Hudi [5], Apache Iceberg [6], and Delta Lake [10] introduce transactional guarantees, schema evolution, upsert/update support, and integration with diverse compute engines. Since 2020, ByteDance has explored adopting these table formats into internal training data infrastructure. However, while these solutions improve certain aspects of data management, they still fall short in

the face of ByteDance's large-scale, complex ML workloads. Key limitations include suboptimal resource utilization, lack of Git-like branch capabilities, inadequate read/write performance, and insufficient integration with training frameworks. Furthermore, the growth of large models amplifies the need for advanced ML data management, particularly for handling diverse data modalities.

Challenges. To fully meet the demands of large-scale ML workloads at ByteDance, we outline four key challenges that must be overcome to establish an effective data management system.

① *Resource-Friendly Storage Formats.* At ByteDance, we manage EB-scale data storage, with massive computational resources engaged in reading and writing data daily for data processing and training. However, popular columnar formats such as Parquet [8] suffer from storage and computing efficiency issues when handling large data, particularly for wide tables with tens of thousands of columns or multimodal data including images and videos. More resource-friendly storage formats are required to mitigate overhead.

② *Efficient Big Metadata Management.* Planning is an essential step in data reading pipelines, responsible for reading, parsing, and processing metadata to obtain file lists. As training data grows in size, metadata becomes big metadata [18]. Existing metadata planning approaches like Iceberg struggle with such metadata volumes, particularly when the metadata includes redundant information. Additionally, isolated experiments are needed for feature research. While Iceberg provides basic snapshot-based branching and tagging functions, ML workloads require enhanced metadata for advanced cross-branch data merging and rebasing capabilities.

③ *Lightweight MOR Update and Upsert.* Many ML scenarios involve frequent data modifications, such as adding new columns in feature engineering, updating advertising conversion data from postbacks, or refreshing multimodal annotation data. Copy-on-Write (COW) and Merge-on-Read (MOR) are two common strategies for data updates. Since COW introduces high writing overhead, which can compromise data timeliness, MOR is preferred at ByteDance. However, MOR strategies of open-source data lakes such as Iceberg often struggle to meet the read and write efficiency requirements under these ML use cases.

④ *Large Model Training Support.* Increasingly diverse large-scale ML scenarios introduce unique requirements for data management and model training. In LRM training, for instance, training samples shift from chronological ordering to grouping by user behavior. Similarly, LMM training involves additional complexities due to its reliance on numerous multimodal data sources. Data management optimizations for large model training are essential to reduce resource consumption and maximize training throughput.

Key Technologies & Contributions. To address these challenges, we propose Magnus, a data management system based on the open-source Apache Iceberg, to accelerate large-scale ML workloads through a series of innovative designs and implements. We summarize the main contributions as follows:

- We introduce columnar Krypton format and Blob format to enhance storage efficiency. Additionally, we integrate inverted indexes and vector indexes directly into the Magnus tables, avoiding data movement to external search systems. (Sec. 3)
- By eliminating redundant fields, sorting by partitions, and constructing indexes of manifest files, we significantly improve the efficiency of large data planning. Besides, Magnus enhances Iceberg's branching and tagging with merge and rebase operations, enabling flexible version control for feature analysis. (Sec. 4)
- We implement a column-level update and a primary key driven upsert based on MOR strategies, while also achieving optimized read performance through native engine enhancements and data reorganization. (Sec. 5)
- To support ByteDance's internal large-scale ML workloads, we carry out targeted adaptations for LRM and LMM training scenarios, applying a dual-table design and a sharding mechanism, respectively. (Sec. 6)
- We experimentally evaluate the storage efficiency, memory footprint, and read/write performance of Magnus. By comparing the performance under production datasets and environments, verifying Magnus's effectiveness large-scale ML scenarios. (Sec. 7)

2 SYSTEM OVERVIEW

Magnus is a unified data management system designed to address the diverse data-related needs in large-scale ML scenarios. As illustrated in Fig. 1, it bridges heterogeneous bottom-layer storage systems (e.g., HDFS, object storage) and various upper-layer ML scenarios (e.g., data ingestion, insight, engineering, and model training). By encapsulating storage backends through a standardized SDK, Magnus abstracts underlying storage complexities and ensures cross-platform compatibility. Magnus primarily contains Magnus Table, Global Lake Service and Multi-Engine Support, each providing specialized functionalities for data storage and processing.

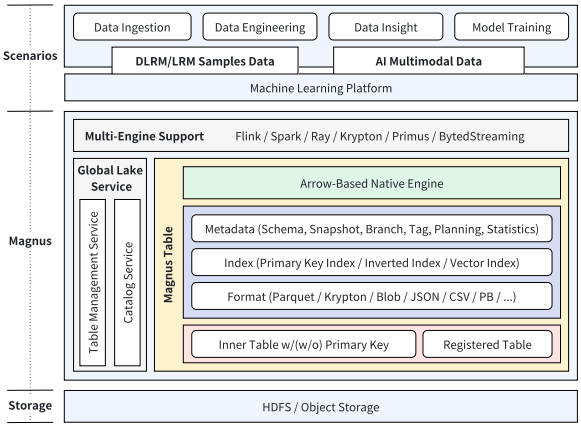


Figure 1: Magnus Architecture

Magnus Table. As the core of Magnus, it unifies both internal and external data through two types of tables:

- Inner table: Includes non-primary key tables supporting standard operations like append and update, and primary key tables enforcing unique key constraints to enable upsert.
- Registered table: Allows for easy integration of external data sources into the Magnus, eliminating data transformation.

Each Magnus Table provides standardized management over formats, indexes, and metadata, ensuring data consistency.

- Format: Natively supports open columnar formats such as Parquet, along with proprietary ML-optimized formats like Krypton

Table 1: Integrated Frameworks and Supported Scenarios of Magnus

Scenario	Engine/Framework	Function Description
Data Ingestion & Data Engineering	Flink [3]	Magnus integrates with Flink to provide unified offline data management from message queues. Flink consumes data from message queues in real-time and writes to Magnus via streaming inserts/upserts.
	Spark [9, 46]	Magnus integrates with Spark to support efficient large-scale batch writing and offline data processing, including read/insert/update/upsert operations. Magnus also enables convenient registration and unified management of external data files.
	Ray [2]	Magnus integrates with Ray to offer complex DAG orchestration and heterogeneous resource scheduling for large models, supporting multimodal data management and read/insert/update/upsert operations.
Data Insight	Krypton [14]	Magnus integrates with Krypton to utilize SQL for data analysis, mining, and visualization, featuring high-performance OLAP and full-text retrieval (including inverted and vector indexes).
Model Training	Primus [40]	Magnus integrates with the self-developed Primus framework, which reads training samples from Magnus tables and supplies data to training frameworks such as PyTorch and TensorFlow for LRM training.
	BytedStreaming	Magnus integrates with BytedStreaming, an internal framework optimized and customized based on open-source Streaming[44], providing flexible and efficient data loading and sharding for LMM pretraining.

format and Blob format. While Parquet is the default ingestion format, non-native formats can be registered directly.

- Index: Employs both primary key indexes and search indexes (including inverted index and vector index). These indexes, combined with metadata-driven file pruning, enable efficient data scanning and retrieval through upper-layer engines.
- Metadata: Maintains critical entities like snapshots, branches, tags, and statistics, supporting granular version control, lineage tracking, and transactional guarantees.

Magnus Table also introduces an Arrow-based native engine to optimize in-memory data processing. It standardizes diverse underlying file formats into Apache Arrow memory format, enabling zero-copy access for upper-layer compute engines and eliminating the overhead of serialization and format transformation.

Global Lake Service. Magnus offers a unified management service to support large-scale data governance for ML workloads including:

- Catalog Service: Organizes all data in Magnus into a hierarchical structure comprising catalog, namespace, and table, enabling systematic management. It allows users to query data through a simplified interface using table names and version identifiers.
- Table Management Service: Provides lifecycle management for tables, supporting version control for historical tracking, role-based access control for security, audit logging for compliance, data time-to-live policies for lifecycle automation, and advanced workflows (e.g., data storage transition, cross-cluster migration).

Multi-Engine Support. Magnus leverages the native engine's core library and catalog service client to provide multi-language SDKs (Java, Python, C++), enabling deep optimization integration with distributed engines and frameworks, including Flink, Spark, Ray, Krypton, Primus, and BytedStreaming. On this basis, Magnus supports ByteDance's end-to-end ML training pipelines across four primary scenarios with corresponding functions as listed in Tab. 1.

Fig.2 illustrates the read and write pipelines of Magnus with integrated engines and frameworks. For writes, engines such as Flink, Spark, or Ray parallelize input into multiple write tasks. Each task is executed independently by the engine's executors, which write data files in supported formats and generate corresponding metadata through Magnus SDK. Once tasks are completed, the driver collects metadata and creates a new snapshot. Metadata files are atomically committed with optimistic concurrency control to prevent conflicts. For reads, Magnus first loads the current table snapshot to plan and generate scan tasks. The scan tasks are distributed across executors, with each retrieving only the relevant columns and records through Magnus's optimized native engine. After the data is loaded, upper-layer engines perform further operations (e.g., filtering, joining, aggregation) to produce the final output.

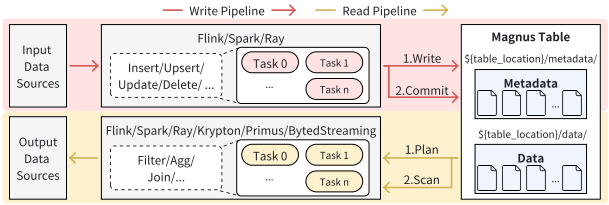


Figure 2: Magnus Write and Read Pipelines

3 STORAGE FORMAT AND INDEX

Magnus follows the hierarchical structure and flexible columnar formats of Iceberg, organizing tables into abstract directory structures on distributed file systems or cloud object storage. Besides supporting open-source formats like Parquet, we introduce novel storage formats, columnar Krypton format [14] and Blob format, to conserve resources and achieve better performance, particularly in scenarios involving wide tables and multimodal data. In addition, we design inverted indexes and vector indexes, enabling simultaneous search capabilities within the same training dataset.

3.1 Data Layout

Magnus supports multiple types of inner tables, including primary key tables and non-primary key tables. Fig. 3 illustrates the data layout of a Magnus table. Each table is stored as a directory containing the following components:

- Data Directory: Stores all data files, partitioned by user-defined partition columns (e.g., date) and further divided into buckets using hashing primary keys. Magnus decouples logical data organization from underlying file formats for seamless integration with various formats like Parquet (.parquet) [8] and Krypton format (.hsap) [14].
- Metadata Directory: Maintains all metadata, including the latest version number, version-specific metadata files, manifest list files, and manifest files. Metadata files contain the table schema, partition specification, and snapshot list. Each snapshot is stored as a manifest list file, which references multiple manifest files. A manifest file stores the data file list along with metadata.

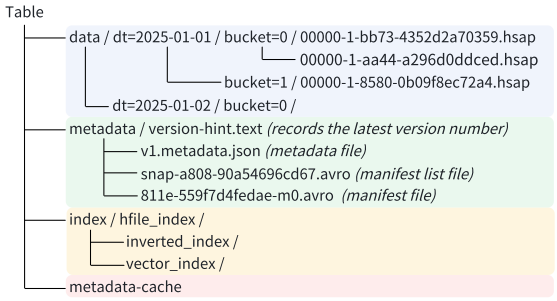


Figure 3: Data Layout of A Magnus Table

- Index Directory: Contains index-related information, including `hfile_index`, `inverted_index` and `vector_index`. The `hfile_index` directory stores HFile index files for primary key tables, while the `inverted_index` and `vector_index` directories store inverted index and vector index files, respectively.
- Metadata Cache: Stores pre-parsed metadata in binary format, enhancing metadata parsing efficiency.

3.2 Columnar Formats

Machine learning models typically rely on specific features (columns) for training. Columnar storage formats significantly improve data loading and preprocessing by allowing reading only the necessary columns from a file, which is particularly beneficial for large-scale datasets. Although Apache Parquet is the most widely used columnar format in the industry, it presents limitations for large, wide tables. A Parquet file is partitioned into row groups, with each row group stored by columns. Consequently, the metadata size and memory consumption of Parquet grow linearly with the number of columns and row groups. Furthermore, before any data can be read, the metadata must be decoded, which incurs substantial CPU overhead in large-scale scenarios.

To solve the problems, we integrate an internally developed columnar format in Krypton [14] with Magnus, which stores data with much less storage space and memory usage than Parquet. This format is structurally organized into three primary components: data region, index region, and file footer. The data region and index region store data pages and indexes, respectively. The file footer contains meta information about the entire file, including the metadata of each column. Consequently, Krypton format maintains a single set of column metadata applicable to all row groups at the file level. Each row group possesses its own specific row group metadata, containing the number of rows and a pointer to each column's metadata. This structure achieves $O(num\_row\_groups + num\_columns)$ memory consumption and much cleaner metadata with the footer represented using Flatbuffers[23] in memory. In contrast, Parquet constructs and maintains a `ColumnChunkMetaData` for every column chunk of every row group, which results in $O(num\_row\_groups \times num\_columns)$ memory consumption.

3.3 Multimodal Data Blob Format

In LMM training scenarios, vast amounts of images, audio, and video data must be efficiently stored and retrieved. Traditional columnar storage formats store multimodal data in columns as binary objects. However, due to the large size of binary data, individual columns become extremely large, limiting the number of records that can be stored in a single file and thereby reducing storage efficiency. Additionally, video frames are typically extracted as images and stored in `List<Binary>` format. During query execution, random access to sub-items requires reading and decoding the entire `List<Binary>`, leading to severe read amplification.

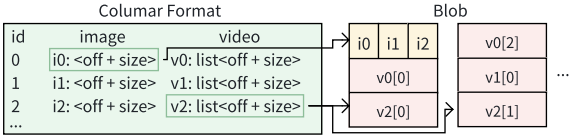


Figure 4: A Blob File Storage Structure Example

Magnus addresses the above challenges using a blob file storage approach. Fig. 4 illustrates the storage structure of blob files. Multimodal binary data is stored separately in blob files using a row-based storage format. The columnar storage format is then used to store only reference metadata, such as the offset and size of each binary object within the blob file. During data retrieval, Magnus first obtains data size and offset from the columnar file and then directly reads the corresponding segment from blob files. For `List<Binary>` data, the position information list is first read from the columnar storage file, sub-items are filtered, and then the required sub-items are read from the blob file. This design enables the columnar files to only store reference information, effectively improving storage efficiency. Moreover, through sub-item filtering and locating based on references of `List<Binary>`, the read amplification in image frame extraction is significantly alleviated.

3.4 Primary Key Index

Efficient data update capabilities are crucial for large-scale ML workloads. Magnus employs a primary key indexing mechanism to optimize update performance for data ingestion. Fig. 5 illustrates the update and read processes based on the primary key index.

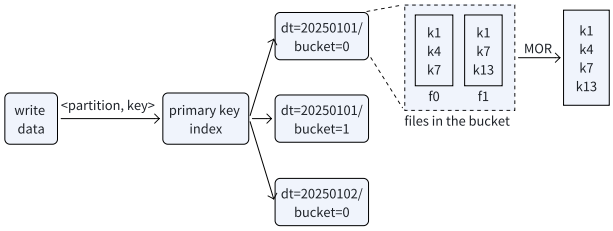


Figure 5: A Primary Key Index Example

The data writing process consists of two stages: primary key indexing and data bucketing. A data record's key is first assigned to a designated data bucket via the primary key index, and then the record is written into the bucket file by a corresponding data writer. Each batch write operation by the computing engine generates a file in the bucket, with records internally sorted by primary key.

During the reading process, Magnus employs multi-way merging strategies to consolidate all bucket files based on primary keys. To

solve ordering conflicts caused by updates to the same primary keys, each written file is assigned a monotonically increasing sequence number. During MOR, the records with higher sequence numbers override those with lower ones, ensuring deterministic resolution of duplicate primary keys. To accommodate diverse use cases, Magnus supports two primary key indexing mechanisms:

- Hash Index: A local primary key indexing mechanism designed for scenarios with fixed partitions. It applies a hash function to compute bucket numbers from primary key values. Each partition directory (e.g., `dt=20250101`) is subdivided into bucket directories (e.g., `bucket=0`), forming a hierarchical structure. This method ensures primary key uniqueness within a partition, but identical keys may exist across different partitions. The hash-based routing requires no external dependencies, reducing computational overhead and ensuring superior performance.
- HFile Index: Ensures global primary key uniqueness across the entire table, particularly useful in dynamic partitioning scenarios. HFile [20], an immutable key-value storage format from Apache HBase, is leveraged to store mappings between primary keys and their corresponding buckets. During indexing, the system first queries HFile to locate the bucket for a given key. If it does not exist in HFile, the system computes the bucket number dynamically and updates the index in HFile.

3.5 Search Index

Multimodal data processing in large model training scenarios expects a single copy of datasets to support both batch training and data search functions, including full-text and vector search. Traditional solutions, such as exporting data to Elasticsearch [35] or external vector databases, introduce significant storage operational overhead due to redundancy. In contrast, Magnus integrates inverted indexing and vector indexing directly into the data lake, enabling cost-efficient and unified data management.

Index construction is implemented through Spark SQL's `ADD INDEX` statement. The following example demonstrates how to create an inverted index on the `content` column:

```
ALTER TABLE magnus.default.t0
  ADD INDEX content_idx(content INVERTED);
```

Each index entry is linked to the original dataset via the corresponding `row_id`, establishing a direct mapping between indexes and the underlying data records. The generated index files are uniformly stored in the index directory in Fig.3, maintaining a one-to-one correspondence with data files. This design simplifies version management by automatically synchronizing index versions with the data files, eliminating the need for separate maintenance.

The query process using indexes contains three phases: planning, task execution, and result aggregation. Here is an example that performs a full-text search using the inverted index:

```
select content, score() from magnus.default.t0
  where match_any(content, 'compute')
order by score() limit 10;
```

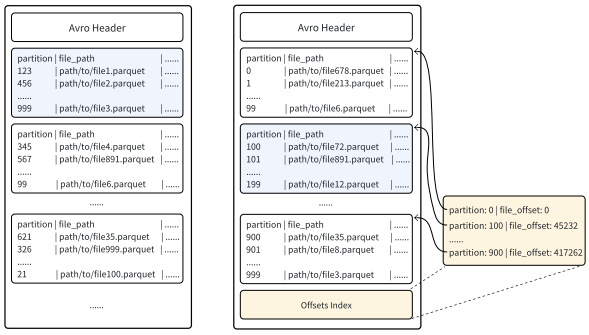
During planning, the query engine Krypton [14] partitions the target table into manageable task splits, each corresponding to one or more Magnus files. During task execution, splits are encapsulated into executable units (i.e., tasks) and distributed to worker nodes. Each task begins by querying the inverted index to retrieve a list of relevant `row_ids` and their BM25 scores (for inverted index) or vector distances (for vector index), filtered by users' search criteria. Leveraging the precomputed `row_ids`-to-data mapping, the task then directly fetches the corresponding records from the underlying Magnus data files. Finally, during result aggregation, Krypton collects partial results from all worker nodes and sorts the unified dataset by BM25 score or vector distance. The top-N entries, ranked by their semantic similarity to the query, are returned to the client.

4 METADATA MANAGEMENT

The metadata layer plays a pivotal role in Magnus, serving as the foundation for advanced data analysis and ML training. Magnus enhances Iceberg metadata to improve the planning performance on wide tables and support lightweight Git-like branching for data isolation and storage saving, with advanced features like merging and rebasing to streamline feature analysis workflows and enhance dataset versioning and traceability.

4.1 Metadata Planning

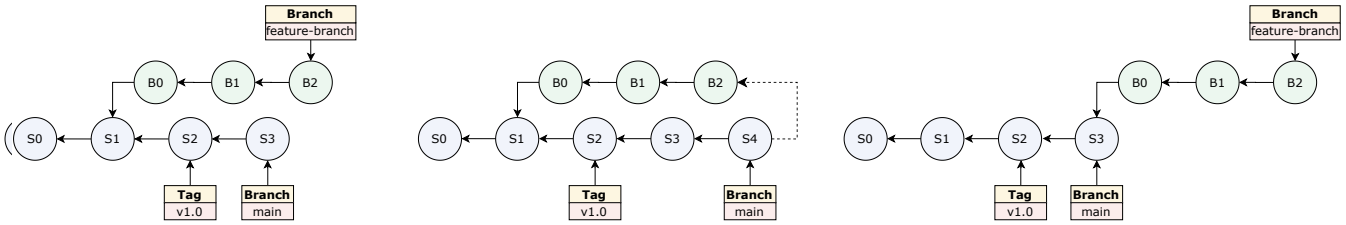
Wide-table queries typically select only necessary feature columns and perform partition scans without additional filters. However, Iceberg persists min-max statistics for each column of each data file in the manifest files by default. Our analysis reveals that such redundant statistics account for 70%-80% of manifest file storage space and parsing overhead, leading to excessive metadata planning latency. To address this, Magnus eliminates redundant metadata such as `lower_bounds` and `upper_bounds` statistics for feature columns during data ingestion, while retaining critical statistics for the normal planning procedure.



(a) Original (b) Optimized

Figure 6: Manifest File Structure

Since each ingestion may generate data across all partitions, the generated entries are randomly distributed within manifest files from different partitions. Consequently, when planning a scan for a specific partition, all manifest files in the snapshot must be fully scanned to filter out non-target partition entries. This results in planning time approaching the full table scan even when reading only a single partition. We address this problem by optimizing the manifest file structure. Before writing a manifest file, we sort its entries by partition value. Meanwhile, we create sparse indexes mapping partitions to file offsets, which record the minimum partition value of each Avro data block and the block's start offset within



(a) Snapshots as nodes, edges as relationships (b) Merging snapshots by a new snapshot S4 (c) Rebasing feature-branch to main branch

Figure 7: Metadata Branching and Tagging Design

the file. When querying the manifest entries of a certain partition, we can skip the blocks that do not contain relevant entries and terminate reading early within blocks based on certain conditions, significantly reducing the manifest file parsing overhead.

For instance, consider a table with 1,000 partitions, where each partition generates a data file during a write operation. The corresponding manifest file is shown in Fig. 6. When querying partition 123, a full scan of the manifest file is required to locate its single relevant entry. With our optimization, the offsets index reveals that the minimum partition value in block 1 is 100, while in block 2, it is 200. Therefore, we only need to read block 1. Inside block 1, due to the ordering of manifest entries, encountering partition 124 within block 1 indicates that no subsequent records belong to partition 123, allowing for an early termination of the reading process.

4.2 Branching and Tagging

Similar to Iceberg, `Magnus` supports lightweight branching and tagging operations, allowing users to create new branches or derive them from existing ones while ensuring snapshot isolation. Fig. 7(a) illustrates the design of branches through a directed acyclic graph, where snapshots serve as nodes and directed edges represent parent relationships. However, in typical ML scenarios, experimental branches often need to integrate validated changes into production pipelines and synchronize with upstream data updates periodically—features not natively supported by Iceberg. `Magnus` further enhances Git-like branching functionality with two critical extensions tailored to feature analysis workflows: merging and rebasing.

- Brach Merge: Enables integration of experimental branches into other branches, as shown in Fig. 7(b). When merging a branch, `Magnus` identifies all manifest files within the source branch and incorporates them into a new snapshot on the target branch. The relative sequence number of the merged manifest files is preserved, maintaining the temporal order of data updates.
- Branch Rebase: Addresses the challenge of experimental branches diverging from upstream changes. When rebasing, `Magnus` picks all the snapshots in the branch and re-commits them on top of a new base branch, effectively resetting the branch's lineage, as shown in Fig. 7(c). During this process, the sequence numbers of the rebased snapshots are recalculated to ensure the highest precedence during MOR.

As a complement to branch management, `Magnus` employs a tagging feature to version datasets for reproducible model training. Tags serve as immutable references to specific snapshots, allowing teams to track data states across experiments. Additionally, `Magnus` integrates commit message logging into write operations, enabling users to audit data evolution. In Spark and Ray workflows, users can attach descriptive commit messages to snapshots, which are stored in the snapshot summary metadata. All these operations above are implemented purely at the metadata layer, thus avoiding redundant data rewrites.

5 HIGH-PERFORMANCE READ AND WRITE

Through carefully designed MOR strategies, `Magnus` enables efficient column-level updates and primary key table upserts, while minimizing write amplification. To mitigate the read amplification typically associated with MOR operations, `Magnus` employs native engine enhancements and data reorganization techniques. These optimizations make `Magnus` particularly well-suited for high-frequency ingestion pipelines, striking a balance between update efficiency and query performance.

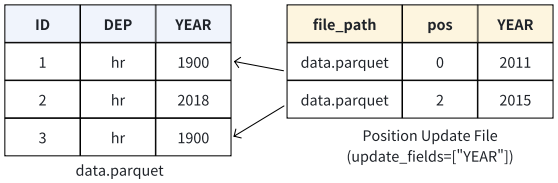
5.1 Column-Level Update

Column-level update means modifying specific columns without altering or rewriting entire rows, which is critical for optimizing storage efficiency and reducing I/O overhead. Traditional data lake frameworks, such as Apache Iceberg, lack native support for column-level updates, often requiring full-row reads and rewrites from the underlying data files. This results in significant read amplification by retrieving unnecessary data and write amplification by rewriting unchanged columns, undermining performance.

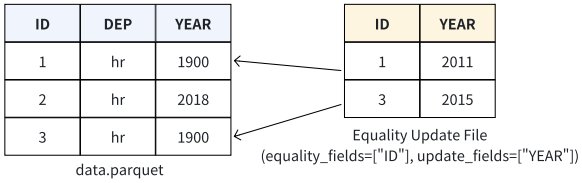
As shown in Fig. 8, `Magnus` provides the capability of column-level updates by implementing two distinct update operations: position update and equality update.

- Position Update: Targets updated rows by physical location (`file_path,pos`), suitable for batch read-modify-write operations. While flexible for complex updates, concurrent modifications to the same data file may cause transaction conflicts requiring retries.
- Equality Update: Identifies rows using user-defined `equality_fields` (e.g., primary keys), designed for streaming write-only operations. It employs MOR strategies to resolve concurrent conflicts, with pre-defined MOR rules required for complex logic such as concatenating lists from multiple transactions.

During the writing process, updates are staged in separate position update files or equality update files without modifying the



(a) Position update



(b) Equality update

Figure 8: Examples of Position Update and Equality Update

base data, and sequence numbers are assigned to record commit orders. Besides, to further enhance performance, Magnus sorts records within each update file by `(file_path,pos)` or `equality_fields` if possible. This enables sort-merge MOR, which is more efficient than hash-join MOR for unordered update files. These optimizations eliminate unnecessary data retrieval and significantly reduce the overhead of column-level updates.

5.2 Primary Key Driven Upsert

Upsert combines insert and update operations, enabling atomic insertion of new records or modification of existing ones based on primary key matching. Open-source frameworks like Apache Iceberg have issues when handling upserts: 1) Streaming upsert requires partition fields in primary keys, fails to ensure global uniqueness, and lacks support for column-level updates. 2) Batch upsert supports global uniqueness, but suffers from high read/write amplification (due to full-row scans rewrites) and transaction retries from concurrent data overlaps. To address these limitations, Magnus proposes a lightweight MOR-based upsert mechanism with conflict-free concurrency strategies for primary key tables.

The design of the primary key table is pivotal to ensuring data consistency and efficient upserts. As outlined in Sec. 3.4, the Magnus primary key tables employ a two-layered partitioning scheme, with user-defined partitions subdivided into buckets via hash operations on the primary key. To enforce primary key uniqueness, the system ensures deterministic routing of upsert records with identical keys to the same bucket within the same partition. This is achieved through two complementary indexing mechanisms: Bucket index (Hash index) and Global index (Hfile index). Within each bucket, upsert records are stored in sequentially numbered data files that include all fields. These files are internally sorted by primary key to optimize read performance and maintain consistency during merge processes. Sequence numbers enforce temporal ordering, ensuring newer data files override older ones. Non-null field values within upsert data files are logically equivalent to `update_field` values in equality update files, applying column-level updates to existing records based on specified MOR strategies.

An upserting process comprises three phases: routing, data writing, and commit. First, the routing phase determines the target partition and bucket for each record. For bucket indexing, the destination is computed directly through hashing. For global indexing, incoming data undergoes a bucket-join operation with the index to identify matching entries, with unmatched records being mapped to their respective partitions and incrementally persisted as HFiles. Next, in the data writing phase, data belonging to identical buckets is routed to the same worker node to minimize file fragmentation. Each worker sorts all internal records by primary key before writing them to data files. Finally, the commit phase is conducted, with an assigned unique sequence number to guarantee serializability and consistency. For concurrent upsert operations, Magnus avoids traditional data-level conflict detection during commits. Instead, it simply creates a new snapshot for each commit at the metadata level, with increasing unique sequence numbers to indicate order. The actual resolution of potential conflicts is deferred to the MOR phase, where data is merged according to sequence numbers as well as selective merging strategies, as described in Sec. 5.3.

5.3 Efficient MOR Strategies with Native Engine

Sec. 5.1 and Sec. 5.2 have introduced MOR-based update and upsert operations, which achieve high-throughput writing, but inevitably lead to a more complex reading process than COW. In this section, we elaborate on our efficient reading process involving phased MOR strategies, as illustrated in Fig. 9. Through carefully designed merging mechanisms and a series of native engine enhancements including predicate pushdown and prebuffering, Magnus minimizes query latency and read amplification.

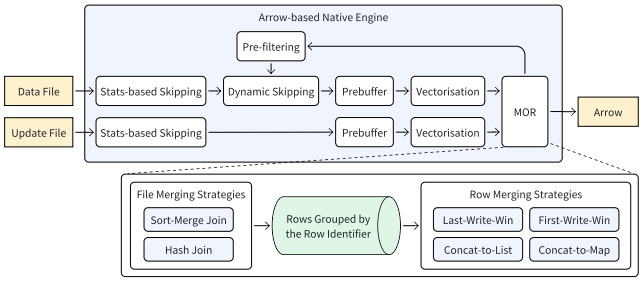


Figure 9: Optimized Reading Process in Magnus Through Arrow-based Native Engine

Reading processes in Magnus consist of two phases: 1) *planning* for task generation and optimization; 2) *scanning* for parallel data retrieval with MOR. During the planning phase, Magnus parses metadata and determines the file lists required for each executable task. For non-primary key tables, each data file is matched with position updates having equal or higher sequence numbers in the same partition to generate scan tasks. For the primary key tables, all data files and update files within each bucket are combined into a single bucket scan task to perform primary key-based merging.

The scanning phase subsequently executes these generated tasks through parallel processing, physically reading all files in one task and merging them. MOR of a scan task consists of two stages. The first stage is file merging that merges all data files and update

files. File merging is performed in two steps. First, each data file is sort-merged with its corresponding position update files. Next, the intermediate results are sort-merged based on the primary key and sequence number. Equality update files are also merged in this step, using different strategies depending on table characteristics: sort merging for sorted equality update files in primary key tables, or hash-join merging in other cases. Rows from all files are finally collected and grouped by their row identifiers. The row identifier is the primary key for primary key tables or (file_path, pos) for non-primary key tables. Within each group, rows are sorted by sequence number to prioritize earlier writes. The second stage is row merging which merges the rows with the same identifier in sequence number order, which produces the final output. When merging rows, multiple conflict resolution rules are configurable to accommodate diverse column-level update requirements, including first-write-win, last-write-win, list-concat, and map-concat.

Magnus further optimizes data reading through its Arrow-based native engine, with two key techniques including predicate pushdown and prebuffering. Predicate pushdown enhances performance by eliminating unnecessary data processing. By pushing predicates down to the native engine, irrelevant data segments can be skipped to avoid unnecessary I/O and deserialization. For Parquet files, the native engine identifies and skips row groups that do not contain any rows satisfying the filtering conditions. Two approaches can determine whether a row group should be skipped. The first is to leverage statistical metadata (e.g., min/max values, bloom filters) stored in the Parquet footer to quickly evaluate filters. The second is to defer full data materialization by initially reading only the columns involved in the predicate, enabling early filtering before accessing the remaining data. Prebuffering complements predicate filtering by asynchronously prefetching row groups from remote storage into local memory. This IO-computation parallelism effectively overlaps data transfer with query processing, thereby reducing the latency associated with data loading.

5.4 Data Reorganization Mechanisms

Poorly designed schemas and frequent write operations can lead to fragmented and suboptimal data layouts, severely impacting both read and write performance. To improve the physical organization of data, Magnus introduces data reorganization mechanisms, primarily including compaction and column reordering.

Frequent column-level updates and upsert operations generate a growing number of incremental data files and global index files over time, increasing the burden on MOR. While traditional compaction strategies—such as those used in Iceberg—attempt to mitigate this by merging all files periodically, they often incur significant resource overhead and operational latency. Magnus provides two configurable compaction strategies to address the degradation of read performance and storage efficiency caused by these fragmented files: *major compaction* and *minor compaction*. Major compaction aggressively merges all incremental files into base data files, maximizing read performance at the cost of higher resource consumption. In contrast, minor compaction adopts a cost-efficient approach that avoids rewriting large historical data files while delivering moderate read performance improvements. It is based on the fact that files with smaller sequence numbers typically represent previously compacted large base files, while those with larger sequence numbers correspond to newly generated small delta files. Thus, this lightweight strategy selectively merges data files and their associated update files within each bucket that possesses sequence numbers exceeding a dynamically determined threshold. Collectively, by applying these two compaction methods, Magnus meets the diverse requirements for resource consumption and reading speed in scenarios with different incremental frequencies.

Wide-table workloads in Magnus often involve selecting hundreds to thousands of columns from tens of thousands available. In columnar formats like Parquet, different column data blocks are stored at disparate file offsets within row groups. For queries that randomly select some columns, this layout leads to frequent small-range random I/O or excessive read amplification when small I/O requests are coalesced, both of which can become performance bottlenecks. To address this structural inefficiency, Magnus implements a column affinity optimization mechanism. It continuously collects column access patterns during query execution and dynamically reorganizes the column layout based on access frequency. Columns that are frequently accessed together are physically co-located in storage to promote sequential I/O and reduce random reads. This spatial optimization significantly alleviates read amplification and improves the performance of frequent queries.

6 LARGE MODEL TRAINING SUPPORT

In this section, we elaborate on the application of Magnus in two distinct large-scale ML training scenarios within our company: LRMs and LMMs. Specifically, for LRM training, we design a novel dual-table structure comprising monthly Main tables and Extra tables, which enables unified user-level feature organization throughout the training process. Regarding LMM training, Magnus provides comprehensive support for multimodal data management and enhanced data sharding and shuffling strategies.

6.1 LRM Training Support

In traditional deep learning recommendation models, data samples are organized as time series and read sequentially by time partitions during training. We observe that samples from the same user across different timestamps often share substantial common prefixes in their historical behavior sequences. To mitigate redundancy, we reorganize the samples by user sequence, aggregating data from the same user within a defined timeframe for user-level training. User sequence samples are stored in Magnus primary key tables, with a redefined primary key `_fs_pk` = `uid:generate_time:uuid`. We consolidate several upstream data sources into a unified table, with distinct data streams differentiated through the `_gr_source` feature column. By redefining the range parameters of partitions, we enforce the storage volume of each bucket to less than 11 GB while accommodating the 60 PB global dataset. It enables the computational reuse of shared behavioral prefixes across different samples, improving training efficiency by over fivefold.

Furthermore, the user sequences in each event feature exhibit infrequent updates and a high degree of repetition. To address storage redundancy of user sequence data, we propose a dual-table structure comprising Main and Extra tables. The Main table stores

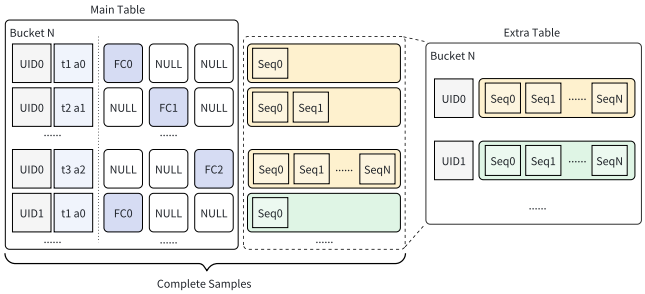


Figure 10: Design of Main-Extra Tables

numerical behavioral features (such as click, dislike, and convert) through monthly partitioning, enabling efficient data retrieval for training. The Extra Table maintains user sequence features separately, preserving only one instance for each user. During model training, sequence segments within specified (BeginTimeStamp, EndTimeStamp) ranges are extracted from the Extra table and then joined with corresponding Main table entries to construct comprehensive features, as illustrated in Fig. 10.

To ensure model freshness and data processing efficiency, the Magnus table undergoes periodic updates and maintenance. User action sequences are incrementally written into the Main table on a daily basis, with multiple data files consolidated via MOR during training. Periodic compaction further optimizes storage. In practice, historical monthly buckets are typically well-compacted, retaining only one to two data files per bucket. Actively updated monthly buckets, which accumulate frequent incremental writes, maintain 15–30 data files depending on compaction frequency. It eliminates redundant storage of identical user sequence features. Given an average of 100 samples per user in our production Main tables, the Main-Extra structure reduces storage needs by approximately 50×.

Based on these designs, we implement a unified framework for organizing LRM training samples in both offline and online scenarios. For offline batch training, samples are organized by users and trained by month. After model deployment, user historical behavior sequences are incrementally updated daily to maintain temporal relevance. In online training, the latest user interactions are dynamically concatenated with historical sequences stored in the system, thereby reconstructing complete time-series sample sequences for real-time model optimization. Additionally, when introducing new user sequence features or updating existing ones, the isolated features stored in the Extra Table avoid large-scale data migrations, minimizing read and write amplification while improving feature backtracking efficiency.

6.2 LMM Training Support

The pretraining of LMMs requires vast datasets, but accessing them directly via HDFS paths often leads to organizational difficulties due to fragmented management. Magnus streamlines dataset management by its unified catalog service, which abstracts complex HDFS paths into a simpler `catalog.database.tablename` identifier for easier access. Additionally, Magnus offers cost-saving capabilities that allow data to migrate seamlessly between different storage media such as SSDs, HDDs, and object storage, based on usage and access patterns. It also supports cross-data-center synchronization of table data and robust data access control mechanisms. These features ensure efficient data management while maintaining consistency across diverse operations and environments.

At ByteDance, we have developed BytedStreaming based on Streaming framework[44] to support distributed large model training. The training data consists of multiple proportionally sampled datasets, with each dataset containing several proportionally sampled streams. During training, different datasets are processed independently. For each dataset, sharding is performed first to collect shards from streams, followed by global shuffling to ensure data randomness. Shards are then evenly partitioned based on the number of samples and data-reading workers. Each worker applies additional shuffling before reading the samples.

The sharding and shuffling operations used to be performed on a combination of HDFS and Redis systems. Users are required to maintain the metadata of all HDFS files manually, and Parquet footers need to be cached in Redis for sharding. We replace the legacy system with Magnus, which autonomously maintains essential metadata for sharding and shuffling without awareness of HDFS physical files. Specifically, enhanced Magnus metadata provides an abstraction of streams and shards for BytedStreaming. A `row_group_counts` field is added to each data file to record the number of rows in each row group. Consequently, a table is represented as a stream, with each row group mapped to a distinct shard. All shards of the stream can be conveniently obtained through the planner, and efficiently read by the Arrow-based native engine.

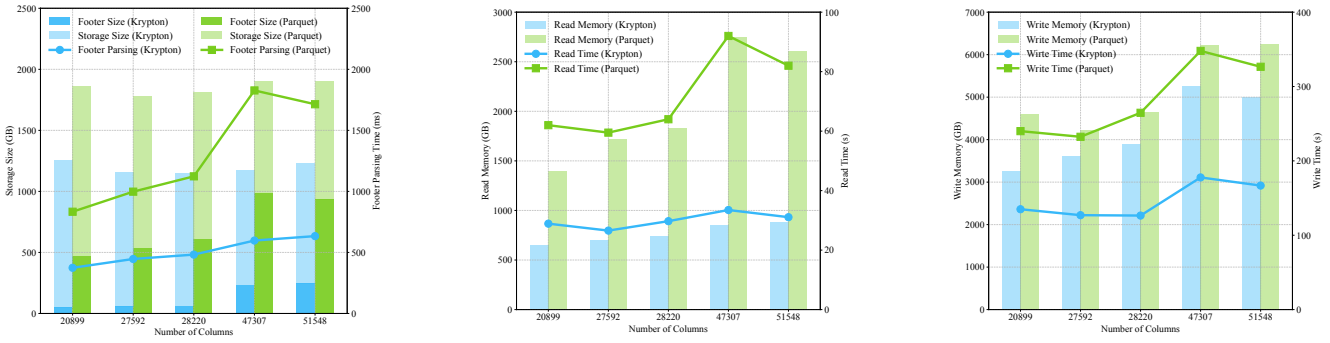
In LMM training, as the number of input datasets increases, the number of shards processed by each node also increases. Due to the considerable size of individual shards in multimodal datasets, frequent out-of-memory errors occur during training, and some tasks even fail to initialize. To mitigate this, we implement row-range planning in Magnus. Each scan task is assigned to an arbitrary region of a data file, enabling dynamic file splitting at a finer granularity based on available node resources and dataset scale. Meanwhile, we design a multi-tier plan cache to address the computational overhead introduced by fine-grained planning. It maintains both remote and local caches to reuse the planning results, achieving comparable startup performance. Moreover, as BytedStreaming retains all scan task objects (i.e., shards) in memory for global shuffling, finer sharding granularity significantly increases the number of shards and thus the memory footprint. We implement Magnus Lite Planner, which compresses scan task objects while preserving essential contexts such as shard identifiers and sample counts for shuffling. The original objects are reconstructed through the compressed context later upon data loading, reducing memory footprint by an order of magnitude without compromising shuffling efficiency.

7 EVALUATION

In this section, we evaluate the performance of Magnus in production scenarios, including the efficiency of storage formats, metadata planning, updates, upserts, and large model training workloads.

7.1 Columnar Format Performance

We evaluate the advantages of Krypton format over Parquet in terms of storage size and memory usage in a wide-table production



(a) Storage Size and Footer Parsing (b) Read Performance (c) Write Performance

Figure 11: Comparison of Krypton and Parquet Columnar Formats in Magnus

environment at ByteDance. Five Parquet files of comparable sizes are randomly selected from CTR table partitions, with each file containing different numbers of columns ranging from 20,899 to 51,548. We convert these Parquet files into Krypton format, ensuring consistency in row content and metadata through validation, and then use these files as datasets. All files are tested under identical hardware and software configurations.

As is shown in Fig. 11, our format exhibits a notably smaller storage size (saving over 30%) and reduced memory footprint during footer parsing and read/write tasks. These improvements stem directly from the optimized handling of redundant metadata structures, which minimizes overhead while preserving data integrity. Furthermore, this optimization contributes to enhanced read and write throughput, underscoring the efficiency of Krypton format in Magnus for large-scale columnar data processing.

7.2 Blob Format Performance

We evaluate Blob format performance for partial access tasks on `List<Binary>` objects, such as video frame extraction. Using a 100 GB Magnus table with a `List<Binary>` column (1024×1 MB binary elements per list), we measure read performance between Blob and Parquet when accessing subsets of elements.

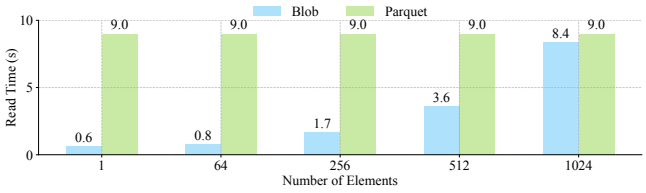


Figure 12: Blob Format Read Performance

Fig. 12 shows that Blob significantly outperforms Parquet when reading partial elements. This stems from Blob's ability to precisely locate target elements using reference information in data files, eliminating read amplification. In contrast, Parquet must read entire lists before filtering desired elements. These results highlight Blob's structural advantage for selective access in binary list structures.

7.3 Metadata Planning Performance

To validate the effectiveness of metadata optimization, we compare the planning phase duration for single-partition queries before and after optimization under identical hardware conditions. The experiments are conducted on a standalone physical machine equipped with one CPU core. Three feature tables with more than 40,000 columns in production are randomly selected, containing 48,666, 142,940,771, and 337,074,100 files, respectively, distributed in 1,024 partitions. As shown in Tab.2, results demonstrate significant improvements in parsing efficiency. The optimized metadata planning Magnus manifest parsing time is 5× to 26× faster than Iceberg, with greater benefits as the number of files increases. This reduction confirms the validity of our metadata in accelerating planning.

Table 2: Metadata Planning Performance

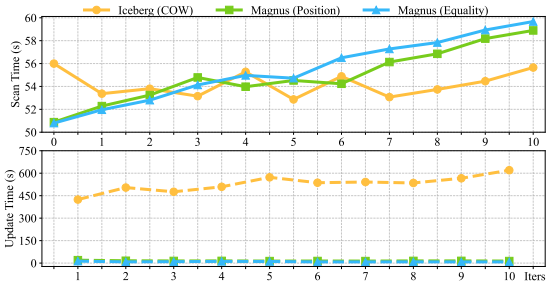
Number of Files	Iceberg	Magnus	Improved
48,666	73s	13.9s	5.25×
142,940,771	2615s	137s	19.09×
337,074,100	6111s	234s	26.12×

7.4 Update Performance

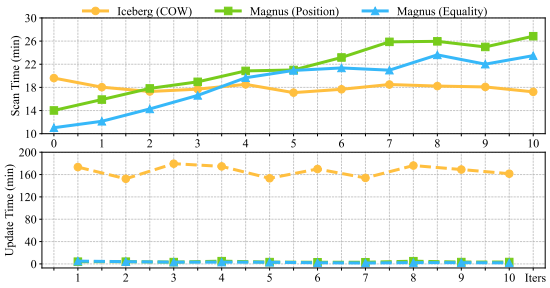
We compare the read and write performance of Magnus's position and equality update strategies with Iceberg using two tables from real-world feature engineering under distinct experimental settings:

- Case 1: A small table (1,800 rows, 200 columns, 116 MB) on a local single-node Spark setup with 1 executor (2 CPU cores).
- Case 2: A large-scale production table (157,598,926 rows, 15,471 columns, 9.4TB) on a distributed Spark cluster with 128 executors (4 CPU cores each).

For both cases, we first perform a full table scan and then conduct 10 iterations of updates on a specified column. In each iteration, all rows of the target column are updated with randomly generated 8KB data, and a full table scan is executed afterward. Due to the severe read amplification of Iceberg MOR under this experimental setup, we compare with Iceberg COW updates to validate the effectiveness of our optimizations. Comparisons with Iceberg MOR are conducted later in Sec. 7.5.

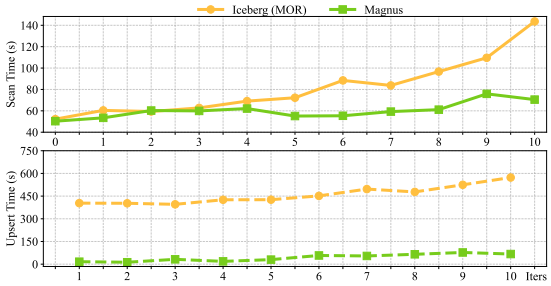


(a) Case 1 (Small Table)

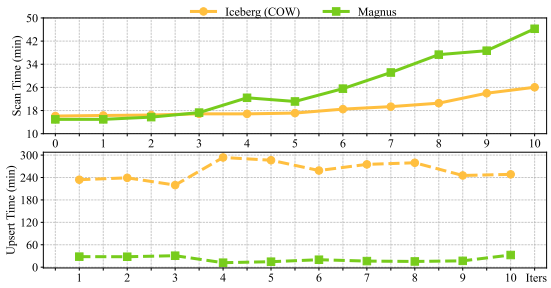


(b) Case 2 (Large Table)

Figure 13: Update Performance Comparison



(a) Case 1 (Small Table)



(b) Case 2 (Large Table)

Figure 14: Upsert Performance Comparison

As shown in Fig. 13, Magnus position updates and equality updates demonstrate significantly superior write performance compared to Iceberg COW updates, because Magnus effectively avoids write amplification via MOR strategies. In terms of read performance, both Magnus position and equality updates achieve optimal efficiency through direct scans, although their read performance gradually degrades with increasing update iterations due to the accumulating overhead introduced by MOR. In contrast, Iceberg maintains stable read performance across iterations by rewriting the entire table during each update. In practical deployments, Magnus limits the number of files within a bounded range through compactions, thereby maintaining read performance that can match or even exceed that of Iceberg's COW strategy. Comparative analysis reveals Magnus's substantial advantage in writing while preserving read performance comparable to Iceberg COW. The result proves the effectiveness of Magnus in workloads with frequent updates.

7.5 Upsert Performance

Fig. 14 evaluates the performance of Magnus upsert strategies with Iceberg upsert using a similar method in the same two cases. We execute 10 iterations of upserts, each updating 5% of existing records and inserting 5% of new data. A full table scan follows each upsert.

For the small case (Fig.14(a)), we compare Magnus MOR upserts with Iceberg MOR upserts. Magnus MOR upserts demonstrate significantly superior write performance over Iceberg, because Iceberg requires joins before upserts to distinguish inserts and updates, whereas Magnus defers this differentiation to the read process. Moreover, Iceberg suffers from write amplification by rewriting entire rows, while Magnus selectively writes only modified column data. The scan time of both Magnus and Iceberg MOR upserts increases with upsert iterations, due to the growth in data volume caused by inserts and the MOR overhead caused by updates. However, the read latency of Magnus increases at a slower rate, as it reads less data and benefits from Magnus's read-oriented optimizations. Collectively, the results confirm the effectiveness of Magnus's upsert strategy, which achieves higher write throughput and lower read latency than Iceberg across all iterations.

In the large case, Iceberg fails to complete MOR upserts within a reasonable time due to severe performance bottlenecks. As shown in Fig. 14(b), Magnus MOR upserts achieve significantly better write performance while maintaining read performance comparable to Iceberg COW upserts. These results further underscore the scalability and efficiency of Magnus in large-scale upsert workloads.

7.6 LRM Training Performance

We evaluate the effects of Magnus in a production LRM training task at ByteDance. We compare resource consumption and performance between two dataset organization strategies: our proposed monthly Main-Extra tables and the original yearly single-table storage.

Table 3: LRM Training Performance

	Yearly Table	Monthly Main-Extra Table
Total Storage	60PB	51.88PB
User Sequence	11PB	2.88PB
Replicas	3500	1400
CPU Cores	16	15
Memory Usage	160GB	70GB
Thoughput	180K instance/s	306K instances/s

As summarized in Tab. 3, the Main-Extra tables demonstrate significant storage efficiency, with the user sequence partition occupying only 2.88GB, compared to 11GB required by the redundant user sequences in the yearly table. Furthermore, our monthly user-level training optimization achieves enhanced read throughput and reduced memory footprint, without requiring additional computational resources such as CPU allocation. These results validate that the Main-Extra tables effectively eliminate data redundancy and improve resource utilization in LRM training workloads.

7.7 LMM Training Performance

We validate the effectiveness of the proposed LLM optimizations through a large vision-language model training job that consumes 300 datasets. Evaluations are deployed on two physical machines, each configured with 118 CPU cores, 2800 GB memory, 8 NVIDIA H800 GPUs, and 2 data-loading workers per GPU. The baseline setup employs the Iceberg Planner with row-group-level sharding.

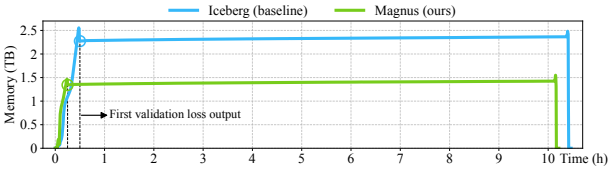


Figure 15: LMM Training Performance

Fig. 15 compares the performance of Magnus and Iceberg in LMM training workloads, focusing on three key metrics: job startup time (from initialization to the first validation loss output), memory footprint, and total training duration. By refining planning granularity to half-row-group level and compressing plan outputs with the Magnus Lite Planner, our optimized Magnus reduces memory consumption by 40%, cuts startup time from 30 minutes to 15 minutes, and shortens training duration slightly by 20 minutes. These results demonstrate the efficiency of applying fine-grained metadata planning for LMM training using Magnus.

8 RELATED WORK

Several studies [25, 28, 48] have extensively analyzed open-source data lake table formats like Apache Hudi [5], Apache Iceberg [6], and Delta Lake [10]. Okolnychyi et al. [36] enhance Apache Iceberg and Apache Spark [9] with efficient petabyte-scale row-level operations. Other data lakes include Deep Lake [26] and StremLake [43]. BigLake [31], an evolution of BigQuery [24, 34], introduces BigLake tables to support fine-grained governance enforcement and performance acceleration. Our work distinguishes itself by introducing substantial enhancements for large-scale ML workloads, including advanced metadata handling and improved scalability.

Columnar formats like Apache Parquet [8] and Apache ORC [7] are widely used in data lakes but struggle with wide-table projections and point queries. Similarly, Doris [19] format lacks $O(1)$ access to page-level data. While Meta's Alpha [12, 51] and ByteDance's earlier Bullion [32] format mitigate wide-table issues, they fall short in point query optimization. Our formats address both wide-table projection and point query inefficiency, offering a more effective solution for multimodal ML workloads. Compared to new formats like Lance [30, 37], Vortex [41] and Nimble [38], our design achieves better simplicity and performance.

Elasticsearch [35] and Milvus [21] provide search capabilities but require external indexing and data duplication. Rottnest [45] advances this by building lightweight indexes directly on data lakes, but it focuses on general search rather than ML pipeline integration or multimodal data. Magnus builds inverted and vector indexes in the data lake, enabling efficient search without additional systems and streamlining ML data retrieval. While Hudi lacks efficient primary key sorting for MOR and Iceberg omits native support for primary key indexes, Magnus implements both HFile indexes and hash primary key indexes, improving update and query efficiency.

Hudi's MOR strategy lacks sorting, and Iceberg does not support column-level updates, limiting write efficiency. For reads, native engines such as Velox [39] and Photon [11] provide high-throughput data queries, but cannot support custom merging strategies and lack corresponding optimizations. Magnus enhances both write and read performance by introducing column-level updates, sort-based merging, efficient indexing, and Arrow-based native engine with techniques including predicate pushdown and prebuffering.

Feature stores such as Hopsworks [16] and Databricks feature store [15] serve ML pipelines by providing a centralized repository for ML features, but they often lack tight integration with data lakes and are limited by scalability issues. Magnus unifies data management and feature storage, supporting LRM and LMM training with optimized table designs and pipeline integration.

9 CONCLUSION

We present Magnus, a holistic data management system designed for large-scale ML workloads. It provides a unified framework to handle massive and multimodal datasets and introduces a series of optimizations on its table format, including resource-friendly storage formats, primary key and search indexes, and metadata enhancement. By enabling lightweight MOR updates and upserts, along with enhanced native engine support, Magnus significantly improves read and write performance, especially in ML feature engineering and model training scenarios. Furthermore, we apply Magnus to LRM and LMM training through training framework integration and unique designs, including Main-Extra tables and fine-grained sharding. In ByteDance's production environment, Magnus has been successfully developed and deployed for more than five years, meeting the challenges of EB-level data scale and providing solid data management capabilities.

ACKNOWLEDGMENTS

We would like to thank all those who contributed to the design and development of Magnus, including Han Qian, Kai Xie, Hanqing Zhao, Yufei Wu, Puke Zhang, Kaiyang Shao, Haoxiang Song, Qianling Li, and Yize Li. We are also grateful to our business partners across the search, advertising, recommendation and Seed teams for their valuable support and collaboration. This work was supported in part by the National Natural Science Foundation of China (62132017, 62421003), "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2024C01167), and ByteDance Research Project (CT20250218122839). Jingyi Ding participated in this work during her research internship at ByteDance.

REFERENCES

[1] Amazon. 2025. Amazon S3. https://aws.amazon.com/s3/ An object storage service offering industry-leading scalability, data availability, security, and performance.

[2] AnyScale. 2025. Ray. https://www.ray.io/ An AI compute engine.

[3] Apache Software Foundation. 2025. Apache Flink. https://flink.apache.org/ A framework and distributed processing engine for stateful computations over unbounded and bounded data streams.

[4] Apache Software Foundation. 2025. Apache Hadoop. https://hadoop.apache.org/ A framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

[5] Apache Software Foundation. 2025. Apache Hudi. https://hudi.apache.org/ An open source data lake platform.

[6] Apache Software Foundation. 2025. Apache Iceberg. https://iceberg.apache.org/ The open table format for analytic datasets.

[7] Apache Software Foundation. 2025. Apache ORC. http://orc.apache.org The smallest, fastest columnar storage for Hadoop workloads.

[8] Apache Software Foundation. 2025. Apache Parquet. https://parquet.apache.org/ An open source, column-oriented data file format designed for efficient data storage and retrieval.

[9] Apache Software Foundation. 2025. Apache Spark. https://spark.apache.org A multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

[10] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment (VLDB)* 13, 12 (2020), 3411–3424. http://www.vldb.org/pvldb/vol13/p3411-armbrust.pdf

[11] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *International Conference on Management of Data (SIGMOD)*. ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[12] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian Sun, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p77-chattopadhyay.pdf

[13] Junyi Chen, Lu Chi, Bingyue Peng, and Zehuan Yuan. 2024. HLLM: Enhancing Sequential Recommendations via Hierarchical Large Language Models for Item and User Modeling. (2024). arXiv:2409.12740 [cs.IR] https://doi.org/10.48550/arXiv.2409.12740

[14] Jianjun Chen, Rui Shi, Heng Chen, Li Zhang, Ruidong Li, Wei Ding, Liya Fan, Hao Wang, Mu Xiong, Yuxiang Chen, Benchao Dong, Kuankuan Guo, Yuanjin Lin, Xiao Liu, Haiyang Shi, Peipei Wang, Zikang Wang, Yemeng Yang, Junda Zhao, Dongyan Zhou, Zhikai Zuo, and Yuming Liang. 2023. Krypton: Real-time Serving and Analytical SQL Engine at ByteDance. *Proceedings of the VLDB Endowment (VLDB)* 16, 12 (2023), 3528–3542. https://www.vldb.org/pvldb/vol16/p3528-chen.pdf

[15] Databricks. 2025. Databricks Feature Store. https://www.databricks.com/product/feature-store The first feature store co-designed with a data platform and MLOps framework.

[16] Javier de la Rúa Martínez, Fabio Buso, Antonios Kouzoupis, Alexandru A. Ormenisan, Salman Niazi, Davit Bzhalava, Kenneth Mak, Victor Jouffrey, Mikael Ronström, Raymond Cunningham, Ralfs Zangis, Dhananjay Mukhedkar, Ayushman Khazanchi, Vladimir Vlassov, and Jim Dowling. 2024. The Hopsworks Feature Store for Machine Learning. In *Companion of International Conference on Management of Data (SIGMOD/PODS)*. ACM, 135–147. https://doi.org/10.1145/3626246.3653389

[17] Chaorui Deng, Deyao Zhu, Kunchang Li, Chenhui Gou, Feng Li, Zeyu Wang, Shu Zhong, Weihao Yu, Xiaonan Nie, Ziang Song, Guang Shi, and Haoqi Fan. 2025. Emerging Properties in Unified Multimodal Pretraining. arXiv:2505.14683 [cs.CV] https://arxiv.org/abs/2505.14683

[18] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata : When Metadata is Big Data. *Proceedings of the VLDB Endowment (VLDB)* 14, 12 (2021), 3083–3095. http://www.vldb.org/pvldb/vol14/p3083-edara.pdf

[19] Apache Software Foundation. 2025. Doris storage file format optimization. https://doris.apache.org/community/design/doris_storage_optimization/

[20] Apache Software Foundation. 2025. HFile. https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/io/hfile/HFile.html File format for hbase.

[21] LF AI & Data Foundation. 2025. Milvus. https://milvus.io/ An open-source vector database built for GenAI applications. Install with pip, perform high-speed searches, and scale to tens of billions of vectors with minimal performance loss.

[22] Yu Gao, Haoyuan Guo, Tuyen Hoang, Weilin Huang, Lu Jiang, Fangyuan Kong, Huixia Li, Jiashi Li, Liang Li, Xiaojie Li, Xunsong Li, Yifu Li, Shanchuan Lin, Zhijie Lin, Jiawei Liu, Shu Liu, Xiaonan Nie, Zhiwu Qing, Yuxi Ren, Li Sun, Zhi Tian, Rui Wang, Sen Wang, Guoqiang Wei, Guohong Wu, Jie Wu, Ruiqi Xia, Fei Xiao, Xuefeng Xiao, Jiangqiao Yan, Ceyuan Yang, Jianchao Yang, Runkai Yang, Tao Yang, Yihang Yang, Zilyu Ye, Xuejiao Zeng, Yan Zeng, Heng Zhang, Yang Zhao, Xiaozheng Zheng, Peihao Zhu, Jiaxin Zou, and Feilong Zuo. 2025. Seedance 1.0: Exploring the Boundaries of Video Generation Models. arXiv:2506.09113 [cs.CV] https://arxiv.org/abs/2506.09113

[23] Google. 2025. FlatBuffers. https://flatbuffers.dev/ An efficient cross platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift.

[24] Google. 2025. Google BigQuery. https://cloud.google.com/bigquery/ The autonomous data to AI platform, automating the entire data life cycle, from ingestion to AI-driven insights.

[25] Rihan Hai, Christos Koutras, Christoph Quix, and Matthias Jarke. 2023. Data Lakes: A Survey of Functions and Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 12 (2023), 12571–12590. https://doi.org/10.1109/TKDE.2023.3270101

[26] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, and Davit Buniatyan. 2022. Deep Lake: A Lakehouse for Deep Learning. arXiv:2209.10785 [cs.DC] https://arxiv.org/abs/2209.10785

[27] Sebastian Hofstätter, Jiecao Chen, Karthik Raman, and Hamed Zamani. 2023. FiD-Light: Efficient and Effective Retrieval-Augmented Text Generation. In *ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 1437–1447. https://doi.org/10.1145/3539618.3591687

[28] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf

[29] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[30] LanceDB. 2025. LanceDB. https://lancedb.com/ A unified data platform designed for multimodal data and built for enterprise scale.

[31] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of International Conference on Management of Data (SIGMOD-/PODS)*. ACM, 334–346. https://doi.org/10.1145/3626246.3653388

[32] Gang Liao, Ye Liu, Jianjun Chen, and Daniel J. Abadi. 2024. Bullion: A Column Store for Machine Learning. arXiv:2404.08901 [cs.DB] https://arxiv.org/abs/2404.08901

[33] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. 2022. Monolith: Real Time Recommendation System with Collisionless Embedding Table. 3303 (2022). https://ceur-ws.org/Vol-3303/paper8.pdf

[34] 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. *Proceedings of the VLDB Endowment (VLDB)* 13, 12 (2020), 3461–3472. http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf

[35] Elastic NV. 2025. Elasticsearch. https://www.elastic.co/elasticsearch Free and open source, distributed, RESTful search engine.

[36] Anton Okolnychyi, Chao Sun, Kazuyuki Tanimura, Russell Spitzer, Ryan Blue, Szehon Ho, Yufei Gu, Vishwanath Lakkundi, and D. B. Tsai. 2024. Petabyte-Scale Row-Level Operations in Data Lakehouses. *Proceedings of the VLDB Endowment (VLDB)* 17, 12 (2024), 4159–4172. https://www.vldb.org/pvldb/vol17/p4159-okolnychyi.pdf

[37] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. arXiv:2504.15247 [cs.DB] https://arxiv.org/abs/2504.15247

[38] Meta Platforms. 2025. The Nimble File Format. https://github.com/facebookincubator/nimble New file format for storage of large columnar datasets.

[39] Meta Platforms. 2025. Velox. https://velox-lib.io/ An open-source composable execution engine for data systems.

[40] Jixi Shan, Xiuqi Huang, Yang Guo, Hongyue Mao, Ho-Pang Hsu, Hang Cheng, Can Wang, Jun Song, Rui Shi, Xiaofeng Gao, Jingwei Xu, Shiru Ren, Jiaxiao Zheng, Hua Huang, Lele Yu, Peng Xu, and Guihai Chen. 2025. Primus: Unified Training System for Large-Scale Deep Learning Recommendation Models. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, 905–922. https:

//www.usenix.org/conference/atc25/presentation/shan-jixi

[41] Spiral. 2025. Vortex. https://github.com/vortex-data/vortex An extensible, state of the art columnar file format.

[42] Shisong Tang, Qing Li, Dingmin Wang, Ci Gao, Wentao Xiao, Dan Zhao, Yong Jiang, Qian Ma, and Aoyang Zhang. 2023. Counterfactual Video Recommendation for Duration Debiasing. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 4894–4903. https://doi.org/10.1145/3580305.3599797

[43] Xin Tang, Chengliang Chai, Dawei Zhao, Haohai Ma, Yong Zheng, Zhenyong Fan, Xin Wu, Jiaquan Zhang, Rui Zhang, Duanshun Li, Yi He, Keji Huang, Guangbin Meng, Yidong Wang, Yuefeng Zhou, Tao Tao, Lirong Jian, Jiwu Shu, Yuping Wang, Ye Yuan, Guoren Wang, and Guoliang Li. 2024. Separation Is for Better Reunion: Data Lake Storage at Huawei. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 5142–5155. https://doi.org/10.1109/ICDE60146.2024.00386

[44] The Mosaic ML Team. 2022. streaming. https://github.com/mosaicml/streaming/ Fast, accurate streaming of training data from cloud storage.

[45] Ziheng Wang, Sasha Krassovsky, Conor Kennedy, Alex Aiken, Weston Pace, Rain Jiang, Huayi Zhang, Chenyu Jiang, and Wei Xu. 2025. Rottnest: Indexing Data Lakes for Search. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1814–1827. https://doi.org/10.1109/ICDE65448.2025.00139

[46] Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, Rui Shi, Xiaofeng Gao, Yuming Liang, Pengwei Zhao, and Guihai Chen. 2024. Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance. *Proceedings of the VLDB Endowment (VLDB)* 17, 12 (2024), 3759–3771. https://doi.org/10.14778/3685800.3685804

[47] Ling Yang, Ye Tian, Bowen Li, Xinchen Zhang, Ke Shen, Yunhai Tong, and Mengdi Wang. 2025. MMaDA: Multimodal Large Diffusion Language Models. arXiv:2505.15809 [cs.CV] https://arxiv.org/abs/2505.15809

[48] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[49] Jiaqi Zhai, Lucy Liao, Xing Liu, Yueming Wang, Rui Li, Xuan Cao, Leon Gao, Zhaojie Gong, Fangda Gu, Jiayuan He, Yinghai Lu, and Yu Shi. 2024. Actions Speak Louder than Words: Trillion-Parameter Sequential Transducers for Generative Recommendations. In *International Conference on Machine Learning (ICML)*. PMLR, Article 2414, 26 pages. https://proceedings.mlr.press/v235/zhai24a.html

[50] Buyun Zhang, Liang Luo, Yuxin Chen, Jade Nie, Xi Liu, Shen Li, Yanli Zhao, Yuchen Hao, Yantao Yao, Ellie Dingqiao Wen, Jongsoo Park, Maxim Naumov, and Wenlin Chen. 2024. Wukong: Towards a Scaling Law for Large-Scale Recommendation. In *International Conference on Machine Learning (ICML)*. PMLR, Article 2455, 14 pages. https://proceedings.mlr.press/v235/zhang24ao.html

[51] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. ACM, 1042–1057. https://doi.org/10.1145/3470496.3533044