ByteHouse: ByteDance's Cloud-Native Data Warehouse for Real-Time Multimodal Data Analytics

Yuxing Han
ByteDance
Shanghai, China
hanyuxing@bytedance.com

Yu Lin
ByteDance
Hangzhou, China
linyu.michael@tiktok.com

Yifeng Dong
ByteDance
Beijing, China
dongyifeng@bytedance.com

Xuanhe Zhou
SJTU
Shanghai, China
zhouxuanhe@sjtu.edu.cn

Xindong Peng
ByteDance
Beijing, China
pengxindong@bytedance.com

Xinhui Tian
ByteDance
Beijing, China
tianxinhui@tiktok.com

Zhiyuan You
ByteDance
Hangzhou, China
youzhiyuan@bytedance.com

Yingzhong Guo
ByteDance
Beijing, China
guoyingzhong.z@bytedance.com

Xi Chen
ByteDance
Shanghai, China
max.chenxi@bytedance.com

Weiping Qu
ByteDance
Shanghai, China
quweiping@bytedance.com

Tao Meng
ByteDance
Hangzhou, China
jingpeng.mt@bytedance.com

Dayue Gao
ByteDance
Beijing, China
gaodayue@bytedance.com

Haoyu Wang
ByteDance
Shanghai, China
wanghaoyu.0428@tiktok.com

Liuxi Wei
ByteDance
Singapore, Singapore
weiliuxi@bytedance.com

Huanchen Zhang
Tsinghua University
Beijing, China
huanchen@tsinghua.edu.cn

Fan Wu
Shanghai Jiao Tong Univ.
Shanghai, China
wu-fan@sjtu.edu.cn

Abstract

With the rapid rise of intelligent data services, modern enterprises increasingly require efficient, multimodal, and cost-effective data analytics infrastructures. However, in ByteDance's production environments, existing systems fall short due to limitations such as I/O-inefficient multimodal storage, inflexible query optimization (e.g., failing to optimize multimodal access patterns), and performance degradation caused by resource disaggregation (e.g., loss of data locality in remote storage). To address these challenges, we introduce ByteHouse (*https://bytehouse.cloud*), a cloud-native data warehouse designed for real-time multimodal data analytics. The storage layer integrates a unified table engine that provides a two-tier logical abstraction and physically consistent layout, SSD-backed cluster-scale cache (CrossCache) that supports shared caching across compute nodes, and virtual file system (NexusFS) that enable efficient local access on compute nodes. The compute layer supports analytical, batch, and incremental execution modes, with tailored optimizations for hybrid queries (e.g., runtime filtering over tiered vector indexes). The control layer coordinates global metadata and transactions, and features an effective optimizer enhanced by historical execution traces and AI-assisted plan selection. Evaluations on internal and standard workloads show that ByteHouse achieves significant efficiency improvement over existing systems.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGMOD Companion '26, Bengaluru, India

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803080

CCS Concepts

• **Information systems** → *Online analytical processing engines*; • **Computing methodologies** → *Machine learning*.

Keywords

OLAP, Data Layout, Query Processing, Multimodal Data, Intelligent Optimization

ACM Reference Format:
Yuxing Han, Yu Lin, Yifeng Dong, Xuanhe Zhou, Xindong Peng, Xinhui Tian, Zhiyuan You, Yingzhong Guo, Xi Chen, Weiping Qu, Tao Meng, Dayue Gao, Haoyu Wang, Liuxi Wei, Huanchen Zhang, and Fan Wu. 2026. ByteHouse: ByteDance's Cloud-Native Data Warehouse for Real-Time Multimodal Data Analytics. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3788853.3803080

1 Introduction

Real-time multimodal data analytics is becoming increasingly important for modern enterprises (see Figure 1). Across ByteDance Cloud, a diverse set of applications, including e-commerce platforms, gaming services, financial systems, and AI-powered applications, operates over large and heterogeneous data modalities. These applications rely on analytical services such as operational dashboards, real-time log analysis, and semantic retrieval over text and image data to obtain timely and reliable insights. For instance, the code assistant Trae [22] requires millisecond-level semantic retrieval across source code, documentation, and execution traces.

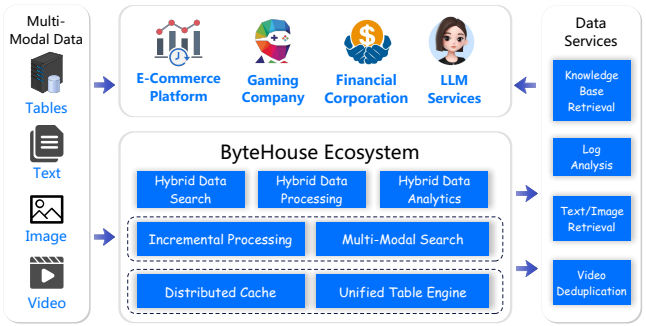


Figure 1: ByteHouse provides real-time multimodal data analytics for over 400 services across ByteDance Cloud.

Existing systems (e.g., data warehouses [1, 7, 18, 30, 37, 59, 73, 75], vector databases [47, 67]) have limitations in meeting these demands. First, many analytical engines [30, 75] offer limited computation optimization for such multimodal workloads. For instance, they lack fine-grained incremental processing at operator level (e.g., delta-aware computation) and the data level (e.g., handling newly ingested rows). Additionally, their query optimizers are typically based on static rule- or cost-driven strategies and lack adaptive mechanisms that optimize evolving query patterns (e.g., text–vector fusion, scalar-constrained vector search) based on historical executions. Existing vector databases [67] and plugin-based solutions [47] also struggle to efficiently optimize such hybrid queries. Second, many cloud-native architectures remain bottlenecked by remote object storage. Although disaggregated storage enables elasticity, it introduces significant latency for sparse or point-access workloads. Even with enhancements like vectorized execution or RDMA-backed memory sharing [37, 73], performance degrades substantially for small or random reads, making real-time responsiveness difficult to sustain. Furthermore, in many cloud-native analytical systems, file-level metadata and auxiliary indexing structures are maintained separately from base data files and stored in distinct object-store locations [18, 52]. Because data files are distributed across object-store partitions and accessed via remote metadata lookups, query processing may incur additional I/O amplification and metadata consistency overhead.

To address these challenges, we propose ByteHouse, a cloud-native analytical data warehouse built upon a shared-storage architecture, which has been refined through *hundreds of application scenarios and tens of thousands of users.* (1) The storage layer provides a unified table engine with a self-describing columnar file format (Sniffer) that natively eliminates external metadata/index dependencies. And ByteHouse provides an SSD-backed distributed cache plane (CrossCache) that can independently scale and close the performance gap of traditional compute–storage separation. Additionally, ByteHouse supports efficient local data access through an alignment-aware filesystem abstraction on each compute node. (2) The compute layer offers three execution modes (i.e., analytic pipeline, staged batch, incremental processing) to handle diverse workloads such as real-time analytical processing. Moreover, we introduce hybrid retrieval operators (e.g., `RANK_FUSION`) for weighted semantic–lexical scoring, scalar-predicate-aware optimizations (e.g., runtime filtering across joined tables), and a tiered vector index design tailored to different service requirements. (3) The control layer manages global metadata, transactions, placement, and scheduling across the distributed architecture. Additionally, the query optimizer supports history-based optimization (HBO), which reuses runtime statistics from past executions to refine selectivity, cardinality, and operator-cost estimates. Building on HBO, ByteHouse incorporates AI models to learn correlations among query structures, data distributions, and runtime behavior to generalize beyond previously observed plan fragments, such as for predicate push-down selection and join side selection.

Contributions. In summary, we make the following contributions:
(1) We propose ByteHouse, a cloud-native, shared-storage data warehouse for real-time multimodal data analytics, supporting high-throughput ingestion and ultra-low latency hybrid queries.
(2) We design a vertically integrated storage layer with a unified table engine, self-describing file format, SSD-based chunk-level caching, and a buffer-managed virtual filesystem that reduces I/O overhead across both the storage and compute nodes.
(3) We propose a unified execution framework combining analytic, batch, and incremental modes, with fusion-based retrieval operators and hybrid query optimization for complicated workloads.
(4) Extensive evaluations on both public benchmarks and ByteDance's internal workloads show that ByteHouse outperforms existing systems on both analytical queries (over 25% lower latency in ClickBench) and multimodal data processing (over 50% higher throughput in Cohere and C4).

2 ByteHouse Architecture

Figure 2 illustrates the architecture of ByteHouse, a cloud-native analytical warehouse built upon a shared-storage design that fully decouples the control, compute, and storage services.

Control Layer. This layer provides global coordination within the distributed framework. The Server handles SQL parsing, semantic analysis, hybrid plan optimization (by history execution statistics and regression models), and distributed scheduling, translating user requests into executable workflows. The Control Services govern global metadata, placement, and system-wide transaction coordination. The Catalog Manager stores versioned metadata in `ByteKV`, exposing snapshot-consistent schemas, partitions, and index definitions across concurrent operations. The Global Transaction Manager issues globally ordered commit timestamps to ensure serializable transactions and consistent snapshot reads. The Daemon Manager orchestrates background maintenance tasks such as compaction and merge scheduling.

Compute Layer. The compute layer hosts a pool of stateless workers operating in a hybrid execution model with three modes: (1) *Analytic Pipeline Mode (APM)* for distributed multi-stage execution with shuffle, gather, and broadcast exchanges; (2) *Staged Batch Mode (SBM)* for long-running ETL with staged retries and shuffle persistence; (3) *Incremental Processing Mode (IPM)* for delta-aware computation with lineage tracking and versioned operator maintenance. All modes share a unified optimizer and runtime to enable seamless transitions. To support multimodal retrieval and analytics, the compute layer provides a spectrum of index structures, including Min-Max, Set, Bloom filters, and vector indexes [28, 40]. ByteHouse additionally introduces hybrid retrieval operators (e.g.,

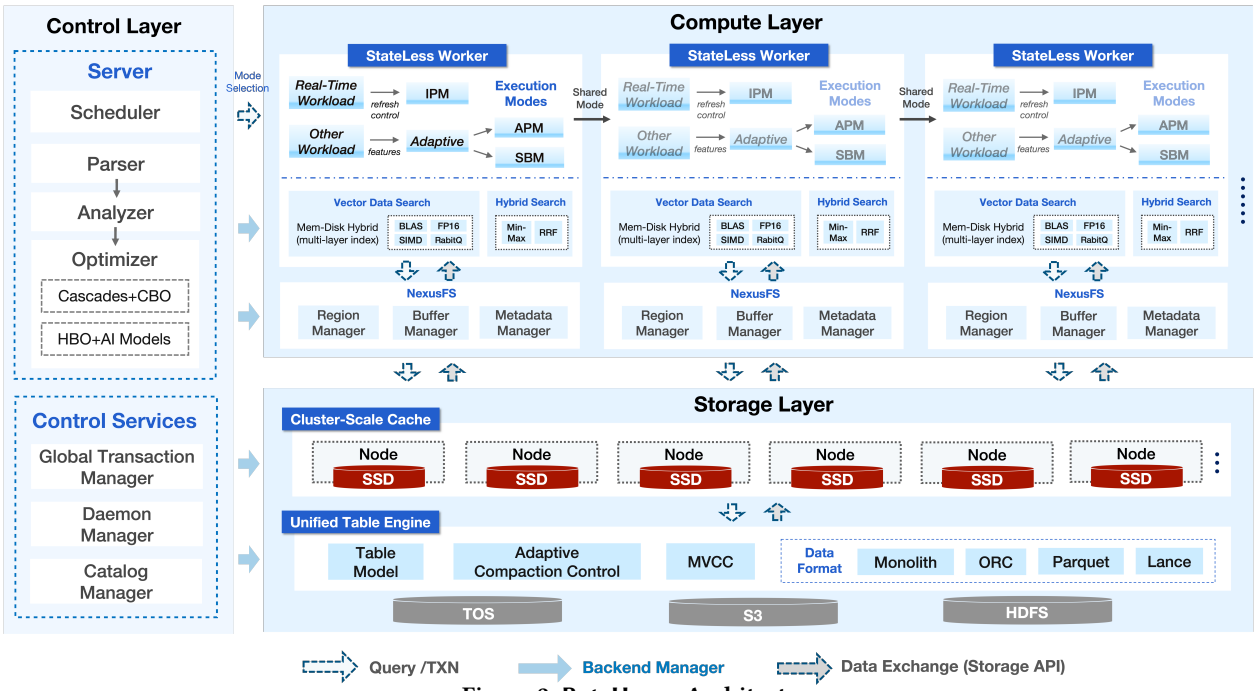


Figure 2: ByteHouse Architecture.

RANK_FUSION), scalar-predicate-aware optimizations (e.g., runtime filters pushed into vector scans), and a tiered vector indexing design tuned for online, near real-time, and cost-sensitive services.

Storage Layer. This layer provides persistent and scalable multimodal data management through three tightly integrated components. (1) The *unified table engine* offers a two-tier logical abstraction (documents–chunks) and a physically consistent layout composed of stable and delta segments. It manages ingestion through a staging–flush write path, maintains snapshot-consistent visibility via MVCC, and supports adaptive compaction and tiered point-lookup resolution essential for analytical workloads. (2) The *distributed cache plane* comprises SSD-backed cache nodes that partition data into fine-grained chunks, apply consistent hashing for balanced placement, and employ prefetching and asynchronous flushing to mitigate remote-storage latency and reduce I/O amplification. (3) The *virtual filesystem abstraction* (NexusFS) unifies access to heterogeneous storage backends, such as TOS (internal object storage), HDFS, and local SSDs, under a single logical namespace. NexusFS provides alignment-aware region management, coordinated metadata lookup, and buffer orchestration to sustain high-throughput read/write paths from compute nodes. Persistent data is stored in columnar formats such as Parquet, ORC, Lance, and the self-describing Sniffer format, enabling efficient hybrid processing across structured, textual, and vector modalities. All compute–storage data exchange is performed through Arrow-based [62] interfaces for zero-copy transfer and high-throughput communication across the disaggregated architecture.

3 ByteHouse Storage

This section presents the storage design of ByteHouse for managing large-scale multimodal data in the cloud. We first introduce the Unified Table Engine, which provides a two-tier logical abstraction and physically consistent layout. We then detail a self-describing file format that co-locates raw data, indexes, and metadata to accelerate operations like point lookups. Next, we describe the distributed cache plane (CrossCache), which mitigates I/O bottlenecks in disaggregated architectures. Finally, we discuss NexusFS, a virtual file system that unifies local disk caching with remote CrossCache access for efficient compute-side reads.

3.1 Unified Table Engine

ByteHouse increasingly serves heterogeneous analytical workloads that span traditional OLAP queries, real-time incremental refresh, and multimodal data analysis. From the perspective of the storage engine, OLAP queries rely on large-scale scans and aggregations over immutable columnar data; incremental pipelines require efficient ingestion and low-latency access to recently updated results; and multimodal workloads often perform feature-level retrieval, such as fetching semantic vectors by primary or sort keys, where point-lookups and fine-grained access latency become critical.

Existing systems typically rely on separate engines to handle these diverse patterns, leading to duplicated metadata and inconsistent data freshness [26]. Instead, ByteHouse introduces a Unified Table Engine that combines these capabilities under a single abstraction. It provides a consistent table model, unified metadata and versioning, and transactional visibility across analytical, incremental, and multimodal workloads.

3.1.1 Table Model Design. ByteHouse introduces a table model that defines how multimodal data are logically represented and physically organized. Each table encapsulates a unified schema that can contain structured attributes, nested fields, and vector columns derived from unstructured modalities. All columns are versioned and tracked by logical descriptors that record data type, encoding,

and statistical summaries. Tables define *primary keys and sort keys*, which together enable efficient range pruning for analytical scans and provide low-latency point lookups for feature retrieval tasks common in multimodal workloads.

(1) Logical Table Design. On top of this table foundation, we introduce a two-tier logical abstraction for multimodal data organization. Each multimodal dataset is represented as a single table comprising a collection of *documents*, where each document denotes a logical entity such as an article, image, or video. Each document is further decomposed into semantically coherent *chunks*, which constitute the fundamental units for embedding-based retrieval and analytical operations. Every record in the table is uniquely identified by a composite key (document_id, chunk_id).

(2) Physical Table Design. Each table is physically composed of *stable segments* and *delta segments* that collectively maintain a consistent version chain. Stable segments store immutable, columnar data optimized for analytical throughput, while delta segments capture recent physical updates, inserts, and compact feature refreshes. Multi-version concurrency control (MVCC) governs the visibility of data across stable and delta segments, ensuring that analytical queries always read from a consistent snapshot even as refresh operations update the underlying data. Background merge and compaction tasks asynchronously incorporate delta segments into stable segments, maintaining high ingestion performance while progressively organizing historical data.

3.1.2 Adaptive Compaction Control. Furthermore, efficient compaction of these delta segments is essential for controlling storage costs and sustaining query performance in large-scale data warehouse storage systems [3, 26]. However, the challenge is that overly frequent compaction amplifies write I/O and inflates storage costs, especially in cloud-native deployments where each write incurs overhead from remote storage, and insufficient compaction leads to segment fragmentation and degraded scan locality over time instead. Thus, ByteHouse introduces an adaptive compaction control mechanism that modulates compaction intensity to resolve the (observed) accumulated delta segments.

Specifically, let N_Δ denote the number of active delta segments, and N^* the equilibrium level that characterizes a balanced compaction state. ByteHouse regulates compaction aggressiveness by a **bounded linear controller**, where the compaction intensity $\alpha \in [0, 1]$ is defined as:

$$\alpha = \min\left(1, \max\left(0, k \cdot \left(\frac{N_\Delta}{N_*} - 1\right)\right)\right), \tag{1}$$

with k controlling the sensitivity to segment accumulation. The coefficient α governs the triggering frequency, merge batch size, and scheduling priority of background compaction tasks. When the number of delta segments is near the equilibrium level, the controller keeps α low, resulting in conservative compaction that avoids redundant compaction work. As N_Δ increases, α increases linearly, progressively relaxing merge thresholds and enlarging compaction batches until the controller saturates at full compaction intensity. This monotonic linear control provides smooth transitions between idle and aggressive compaction phases, preventing oscillation and maintaining steady ingestion performance.

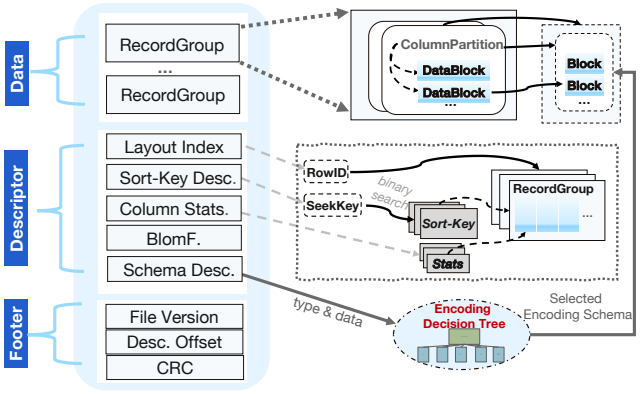


Figure 3: Sniffer File Format.

3.1.3 Tiered Storage Pipeline. To efficiently handle small and frequent row-level ingestion without sacrificing columnar efficiency, ByteHouse employs a two-stage write pipeline that decouples transient ingestion from durable columnar persistence [6, 34].

(1) Staging in Key-Value Store. Incoming updates are first written to a lightweight *staging area*, supported by a distributed key-value store at ByteDance, `ByteKV`. Each write is assigned a transaction identifier and recorded in a Write-ahead Log to ensure durability and atomicity. The staging area serves as a short-lived buffer that accumulates recent changes in row-oriented format.

(2) Flush to Columnar Storage. Once the data size or retention time exceeds predefined thresholds, ByteHouse flushes the buffered data to columnar storage, reorganizing it into compressed column segments. During this flush process, schema evolution and version visibility are preserved, allowing newly persisted data to integrate seamlessly with existing analytical files.

To support low-latency point access, ByteHouse resolves lookups across both staging and columnar layers under snapshot isolation. The engine first probes the transactionally consistent staging store for the most recent visible version of a tuple and falls back to columnar data parts when necessary, leveraging part-level pruning metadata to minimize I/O. This tiered resolution follows the general separation of write- and read-optimized storage and ensures consistent visibility without requiring full-part scans.

3.2 Self-Describing File Format

Next we introduce the file format in ByteHouse. In traditional column stores, column data and auxiliary indexes are maintained as separate structures. While this organization is effective for large sequential scans, it forces point and small-range queries to open additional files, issue seeks, and perform metadata lookups, leading to substantial I/O amplification and metadata overhead in lookup-intensive or hybrid analytical workloads. Furthermore, the common practice of representing `Array` columns as flattened offset-value pairs lacks efficient support for modern multimodel data, such as high-dimensional embedding vectors.

To overcome these limitations, ByteHouse adopts a self-describing file format (Sniffer) that co-locates data blocks with their file-level indexes and schema metadata within a single cohesive layout. This way, Sniffer eliminates external index and metadata files and enables self-contained data access with minimal I/O overhead.

3.2.1 File Organization. Sniffer is composed of Data, Descriptor, Footer regions in a hierarchical structure (see Figure 3).

(1) Data Region stores compressed columnar data arranged in three levels: (i) A *RecordGroup* corresponds to a batch of rows that share the same schema and partitioning key, analogous to a row group in Parquet or a stripe in ORC. (ii) Each record group contains multiple *ColumnPartitions*, each representing a contiguous subset of a column's values within the group. (iii) Every column partition consists of one or more *DataBlocks*, the smallest physical units that store compressed, type-specific blocks of values.

(2) Descriptor Region maintains all structural metadata required to interpret the Data Region efficiently: (i) The *Layout Index* maps RecordGroups and ColumnPartitions to their physical DataBlocks and file offsets; (ii) The *Sort-Key Descriptor* records ordering information for binary search and seek-key resolution; (iii) The *Column Statistics* store block-level summaries such as min/max and null counts for predicate pruning; (iv) The *Bloom Filter* provides coarse-grained filtering for selective queries; (v) The *Schema Descriptor* defines the logical schema, data types, and encoding metadata (e.g., compression codec) used in the file. Note, all metadata are file-scoped, which can be co-located with the storage layer within the file or maintained in a separate search acceleration layer [48].

(3) Footer Region serves as the global directory and integrity anchor of the file [66, 74]. It records the physical descriptor offsets, the file version, and the Cyclic Redundancy Check (CRC) values across the data and descriptor regions. This design allows the reader to locate and validate all descriptors with a single footer access, enabling complete reconstruction of the file layout without relying on external catalogs. Besides, the footer preserves version metadata to ensure backward compatibility and supports hierarchical integrity verification across regions.

Built upon this file format, we can accelerate fine-grained access to compressed columnar data by minimizing random I/O. At query time, the Layout Index, Sort-Key Descriptor, and Column Statistics collectively provide precise navigation over the hierarchical storage structure, allowing the engine to identify and load only the data blocks relevant to a given predicate, including key-based lookups. For point lookups, the engine first identifies the target RecordGroup via the Sort-Key Descriptor, then consults the Layout Index to fetch the precise DataBlock offsets. Because both descriptors are memory-resident, each lookup typically requires only one metadata seek and one block read, achieving ***microsecond-level*** latency even under high analytical concurrency.

3.2.2 Column Encoding Schemes. Additionally, the column encoding methods are critical for the lookup efficiency (e.g., I/O, decoding overhead). Thus, Sniffer dynamically encodes each column according to its data type and statistical distribution [33]. During write time, ByteHouse samples each column and selects the encoding scheme that minimizes both storage footprint and decoding cost. Sniffer integrates several state-of-the-art encoding schemes for different data characteristics: Frame-of-Reference (FOR) combined with Bitpacking [36] with narrow value ranges; Run-Length Encoding (RLE) for low-cardinality columns with frequent repetitions; Dictionary Encoding for categorical strings and enumerations, reducing both size and comparison overhead during predicate evaluation; Finite-State Symbol Table (FSST) [10] for general text or high-entropy string fields; and Adaptive Losses Floating Point (ALP) [2] for floating-point columns, where precision can be adaptively reduced without compromising analytical accuracy.

3.2.3 Format Optimization for Vector Data. In general column stores [18, 30, 52], vectors are stored as flattened `Array` columns in the form of two companion buffers, i.e., an offset array indicating element boundaries and a value array storing all elements contiguously. While compact, this organization introduces several inefficiencies when applied to nested or vectorized data. First, pooling all vector elements into a single value buffer prevents the system from maintaining per-vector statistics or applying compression schemes that operate at vector-block granularity. Second, sparse or variable-length vectors must be represented using placeholder offsets or filler values, resulting in unnecessary storage space waste.

To address this, Sniffer adopts the Length-and-Presence (L&P) representation [44] instead of the repetition/definition encoding used by Parquet [43]. L&P attaches lightweight length and presence metadata to each vector, which enables Sniffer to isolate every embedding as an independent physical unit. This structure enables the system to maintain vector-level statistics, such as value ranges, norms, and nullness, within the Descriptor Region. Besides, it naturally captures sparse or variable-length embeddings without requiring padding or auxiliary offset structures. Therefore, storage cost scales with actual content rather than with the declared dimensionality. Since each vector is stored as a contiguous slice, block-oriented encodings such as FOR or ALP can be applied independently to each vector block. This improves locality and enables SIMD-efficient decoding for dense numeric vectors.

3.3 Cluster-Scale Caching on Storage Side

In cloud-native architectures, decoupling elastic compute from shared storage exposes a substantial performance gap when analytical workloads operate on cold or partially cached data. As a result, query execution often becomes I/O-bound, with remote storage accesses introducing high latency. These factors amplify tail stalls and frequently dominate end-to-end analytical response time. Production traces from ByteHouse deployments show that small-block I/O can vary by more than an order of magnitude across percentiles, with over 70% of execution time spent on cold reads in I/O-intensive workloads. As the cluster scales, read amplification and tail latency increasingly constrain analytical performance.

Distributed Cache Design. We design CrossCache, a distributed caching plane that mitigates remote-I/O bottlenecks by serving hot data from nearby nodes. CrossCache is composed of two main parts: (1) Cache Coordinators (CCs) that manage the global namespace and metadata, and (2) Cache Nodes (CNs) that provide local SSD-based caching and interact directly with different storage backends. Each cached file is divided into fixed-size blocks (12 MB by default), with block metadata and placement information maintained by the CNs. CNs handle both read and write operations. For read paths, when a file is first accessed, the CN checks whether its metadata exists in the cache namespace; if absent, it retrieves the metadata from the backend, partitions the file into blocks, and registers them

for subsequent access. For write paths, CNs buffer new data locally and coordinate block uploads once the buffers are filled. Small blocks are merged into unified files for distributed file systems or concatenated into larger objects for object stores. CNs periodically report block mappings and metadata to the CCs, which consolidate these updates to maintain global consistency across the cluster.

Fine-grained Chunk Caching. To further mitigate I/O amplification and improve cache effectiveness under highly fragmented analytical access patterns, CrossCache adopts a fine-grained chunk caching strategy. Each cached data block is further divided into 4 MB chunks, which are indexed in memory at each CN. Contiguous chunks are appended sequentially to the SSD-resident block file, whereas non-contiguous chunks are temporarily buffered until they can be coalesced into a continuous range. To address the limited write concurrency of storage backends such as HDFS (where each file admits only a single writer), CrossCache employs a ***parallel flushing*** mechanism. Multiple CNs upload their completed block files concurrently as temporary objects, fully utilizing available network and I/O bandwidth. A lightweight concat operation later merges these objects into a single backend file. This design substantially improves write throughput and reduces persistence latency under data-intensive workloads.

3.4 File System Abstraction on Compute Side

While CrossCache enables shared caching across compute nodes, each compute worker still requires an efficient local access path to interface with it. In this context, a data fragment refers to a logical sub-range of a file requested by the execution engine.

In many existing local caching mechanisms, these fragments are handled independently at arbitrary byte offsets [9]. When such requests do not align with the underlying storage block boundaries (e.g., SSD page size or object-store block size), the system must perform unaligned I/O, meaning that the requested byte range spans multiple physical blocks or requires partial-block reads. Frequent unaligned small I/Os degrade throughput, waste SSD and network bandwidth, and undermine the effectiveness of prefetching, especially in scan-heavy analytical workloads.

NexusFS bridges this gap by introducing an alignment-aware file-system abstraction that unifies local disk caching and remote CrossCache access. It enforces end-to-end alignment across the remote data layout, on-disk cache regions, and in-memory buffers, ensuring that data is stored and accessed in coalesced and sequential units across all layers. Internally, NexusFS consists of three tightly integrated components: a region manager, a buffer manager, and a metadata manager. The ***region manager*** partitions local disk space into fixed-size regions, typically 1 MB each, which are further divided into data segments that serve as the basic units of caching and I/O scheduling. It maintains a global index mapping logical file offsets to cached segment locations and reclaims space through a FIFO-based eviction policy [72]. The ***buffer manager*** governs in-memory caching through a fixed-size pool of segment-aligned buffers, applying a second-chance replacement policy to balance reuse and concurrency. When a segment is retained for query processing, it can be directly exposed to the execution pipeline, enabling *zero-copy* reads. The ***metadata manager*** maintains a hierarchical namespace for all files accessed through NexusFS, using a two-level hash hierarchy to track cached segments efficiently [76]. This structure supports constant-time lookups while minimizing memory overhead by selectively serializing inactive entries.

4 Execution Engine

ByteHouse provides three execution modes: the Analytical Pipeline Mode (APM) for low-latency interactive analytics, the Staged Batch Mode (SBM) for high-throughput offline pipelines, and the Incremental Processing Mode (IPM) for continuously ingesting new data. APM and SBM are coordinated by a mode-adaptive execution framework that uses an AI-assisted selector to extract runtime features, predict resource demand, and route each query to the more efficient mode. For IPM, ByteHouse incorporates a rule-based refresh controller that adjusts the refresh interval using a lightweight stabilization policy to balance freshness and maintenance cost.

4.1 Execution Modes

This subsection presents the three execution modes of ByteHouse.

4.1.1 Analytic Pipeline Mode. To accelerate analytical workloads, ByteHouse supports an Analytic Pipeline Mode (APM) that optimizes query planning and resource allocation for low-latency analytical processing. At its core, APM adopts a vectorized, pipeline-parallel execution model that processes columnar data in CPU-efficient batches, achieving high instruction throughput with minimal scheduling overhead. APM incorporates several operator-level optimizations to accelerate core analytical primitives.

For aggregations, it employs an *adaptive* mechanism that samples early input to estimate grouping-key cardinality and the reduction ratio, dynamically choosing between partial aggregation or direct shuffling followed by merge aggregation to avoid unnecessary local grouping and redundant merge computation. For joins, APM applies **runtime filters** generated by build-side operators and pushes them down to probe-side scans, enabling early elimination of non-matching join keys and significantly reducing both probe-side data volume and the irregular, non-sequential access patterns that typically arise during join probing.

To maintain stable pipeline execution under high concurrency, APM integrates a **credit-based flow control** scheme in which each downstream operator grants a bounded number of processing credits to its upstream producers. This regulates data production and buffer occupancy, preventing excessive queuing and sustaining steady pipeline throughput. APM also supports an **ordered consumption** mechanism that preserves the order of intermediate results when the upstream operator already produces ordered output. The consumer incrementally merges incoming ordered segments as they arrive, avoiding a full materialization-and-sort step and reducing overall query latency.

4.1.2 Staged Batch Mode. While APM targets interactive and low-latency analytical workloads, many production scenarios, such as large-scale ETL jobs and LLM-oriented data normalization pipelines, require long-running and throughput-oriented execution. To support these workloads, ByteHouse provides a Staged Batch Mode (SBM), a stage-based execution framework built for stability, recoverability, and sustained throughput. SBM divides a query plan into a sequence of stages separated by *exchange* boundaries. Each stage is decomposed into parallel tasks that operate on disjoint

data partitions, and a stage begins execution only after all of its upstream dependencies have completed. Within a stage, tasks can **materialize intermediate results** to temporary storage, allowing downstream tasks to retrieve their inputs from local or remote spill files based on data locality. These materialized outputs act as lightweight checkpoints and enable **task-level retries**, where failed tasks can be relaunched without restarting the entire stage, significantly reducing recovery time and operational overhead. SBM further supports **elastic parallelism**: input partitions assigned to a compute worker can be processed in multiple batches, allowing the system to flexibly increase task parallelism while keeping each task's memory footprint small.

4.1.3 Incremental Processing Mode. To support near real-time analytical workloads, IPM maintains continuously updated query results as base tables evolve. Unlike APM and SBM, which execute queries by recomputing their inputs in a full scan or a stage-based batch pipeline, IPM maintains operator state across refresh cycles and evaluates only the data deltas produced since the previous materialization point. This incremental model significantly reduces computation cost and end-to-end latency while preserving transactional consistency [4, 11, 58, 68]. Realizing IPM requires coordinated changes across the query processing stack.

SQL Semantics and Triggering: IPM remains fully compatible with standard syntax and extends DML statements with *refresh interval* annotations that specify the desired maintenance frequency. Shorter intervals provide fresher results at higher compute cost, whereas longer intervals reduce system load while maintaining consistency guarantees.

Row-Level Lineage and Delta Model: IPM introduces *row-level lineage tracking* to support deterministic and fine-grained incremental maintenance. Unlike simple modification flags that merely indicate whether a row has changed, ByteHouse associates each tuple with two immutable metadata fields: i) `tuple_key`, which uniquely identifies the logical identity of a row across versions; ii) `update_seq`, a monotonically increasing sequence number that records the version order of updates for that logical row.

During incremental execution, operators consume streams of delta tuples annotated with this lineage metadata. The lineage information is systematically propagated along the operator pipeline to preserve traceability. For example, in an inner join, the output tuple inherits a composite lineage derived from the participating base tuples. When a base tuple is updated or deleted, the engine uses its `tuple_key` to locate and retract previously materialized join results associated with earlier `update_seq` values. This ensures correctness under incremental recomputation without requiring full re-scans of base tables.

Operator-Level Delta Execution: All incremental operators consume and emit delta records of the form <`tuple_key`, `update_seq`, op>, where op $\in \{insert, delete\}$. A logical update generates a delete delta for the previous version, followed by an insert delta for the new version. Upon receiving a delete delta, an operator locates previously materialized state entries (e.g., stored in state tables) derived from the given `tuple_key`, removes their contribution, and propagates a corresponding delete delta downstream. This uniform delta protocol ensures compositional retraction across operators.

For aggregation, IPM maintains a state table keyed by grouping attributes, where each entry materializes the current aggregate state, including partial aggregates and any derived results. Incoming deltas trigger state lookups and either apply new values or retract obsolete ones, while groups whose aggregate values (e.g., `COUNT`) drop to zero are eliminated through lightweight deletions. Aggregate maintenance is specialized according to algebraic properties. For natively retractable aggregates, such as `COUNT`, `SUM`, and `AVG`, IPM adopts fully incremental implementations. Insert deltas are accumulated into the maintained state, whereas delete deltas are subtracted from their prior contributions. In contrast, aggregates that do not efficiently support retraction under arbitrary update streams, such as `MIN`, `MAX`, are handled using a fallback strategy. IPM retains the original per-group values within the operator state to enable recomputation confined to the affected group. When a delete delta invalidates the current aggregate result, for example, by removing the current minimum, the operator recomputes the aggregate over the affected group. This design preserves correctness while confining recomputation to impacted groups, trading additional memory overhead for bounded recomputation cost.

For inner joins, the optimizer rewrites each join into three delta subqueries that combine the left delta with the right base, the right delta with the left base, and both deltas together. These subqueries execute in parallel over versioned inputs managed by the GTM to ensure snapshot consistency. Their outputs are then unified through a lineage-based reconciliation step that merges overlapping tuples by `tuple_key` and modification sequence.

For left or right outer joins, IPM augments the plan with correction terms that maintain the correct semantics of null extension. The system tracks whether each row currently has at least one matching partner and emits updates whenever this match status changes. When an unmatched row gains a match, its null extended output is withdrawn, and the matched rows are emitted; when the final match disappears, the matched rows are retracted, and a single null extended row is produced.

4.2 Adaptive Execution and Refresh Control

To balance responsiveness, throughput, and data freshness, ByteHouse employs a learning-based mode selector that adapts query execution to workload characteristics, and a rule-based refresh controller that schedules incremental maintenance for IPM.

4.2.1 Adaptive Mode Selection Between APM/SBM. The learning-based mode selection framework performs predictive mode selection using a lightweight regression model. Upon query submission, a feature extractor parses the SQL statement and derives a composite query feature vector from the query plan. The vector consists of three parts: 1) query-level features, such as query length and number of views, 2) access-pattern features, including a one-hot encoding of referenced tables, and 3) plan-structural features obtained by bottom-up traversal of plan nodes, where each node is represented by an M-dimensional structural vector. This feature vector is fed into a regression model trained from historical executions. The model jointly predicts query latency, CPU usage, and memory demands, which are then mapped to mode classes (i.e., APM or SBM) using *multi-dimensional percentile* thresholds derived from the cluster's recent workload statistics. These thresholds are

continuously recalibrated to capture runtime variations in resource contention and query concurrency, ensuring balanced mode selection between responsiveness and throughput. The prediction model is periodically retrained with recent execution statistics to maintain accurate and adaptive resource estimation.

4.2.2 Rule-based Refresh Control. Currently, IPM is explicitly triggered by users when continuous or periodic data maintenance is required. Automatically determining an appropriate refresh interval is crucial for incremental processing, as manually configuring a fixed refresh interval can lead to either excessive recomputation or stale data under varying workloads. To address this, ByteHouse employs a window-based stabilization policy that smooths transient fluctuations in maintenance cost and ensures convergence toward a steady refresh frequency. Instead of relying solely on the most recent observation, the policy maintains a sliding window of execution times from recent refresh rounds, denoted by $T_1, T_2, ..., T_N$. It estimates the average incremental maintenance cost as:

$$T_{avg} = \frac{1}{N} \sum_{i=1}^{N} T_i, \tag{2}$$

where N is typically small (e.g., 3-5) to balance stability and responsiveness. The next refresh interval is computed as:

$$\Delta t = \min\left(\max\left(k \times T_{\text{last}}, \Delta t_{\min}\right), \Delta t_{\max}(U)\right), \tag{3}$$

which explicitly restricts the interval to lie within a lower bound Δt_{min} and an upper bound $\Delta t_{max}(U)$. The scaling factor k prevents overly frequent maintenance, while Δt_{min} protects the system from thrashing. The upper bound $\Delta t_{max}(U)$ adapts to real-time cluster utilization U: ByteHouse enlarges it under high CPU or I/O load to reduce contention, and contracts it under low load to improve freshness. A simple linear model is used in practice:

$$\Delta t_{max}(U) = \Delta t_{base} \times (1 + \alpha U), \tag{4}$$

where Δt_{base} denotes the nominal upper bound under idle conditions, and α controls the sensitivity to load variation. By averaging recent refresh costs and adapting to real-time cluster utilization, this policy absorbs transient spikes, prevents runaway growth of Δt, and converges to a stable refresh frequency that balances data freshness with overall system efficiency.

5 Query Optimization

This section describes ByteHouse's query optimization techniques. Section 5.1 describes the optimizer architecture and its Cascades-based rewrite and enumeration framework, while Section 5.2 introduces AI-driven methods for predicate pushdown selection and join side selection.

5.1 Cascades Optimizer

ByteHouse's query optimizer builds upon the Cascades framework [24, 25, 57] and extends it with a unified cost-based model (CBO) that reasons about partitioning, sorting, and grouping properties during query optimization [77]. Each operator is annotated with structural metadata describing data distribution and ordering, allowing the optimizer to derive and enforce these properties *consistently* throughout plan enumeration. To efficiently explore the search space of bushy join trees, ByteHouse adopts the branch partitioning top-down enumeration algorithm [20], which enumerates join combinations in constant time and eliminates the complex data structures required by earlier approaches. The optimizer further incorporates a Magic Set rewriting [53, 54] to enhance predicate selectivity propagation. By replicating selective subplans into upstream operators, this transformation enables early filtering and significantly reduces intermediate results. ByteHouse also employs a cost-based optimization for Common Table Expressions (CTEs), where the optimizer decides adaptively whether each CTE should be inlined, shared, or materialized based on its contextual reuse and query semantics [19, 56].

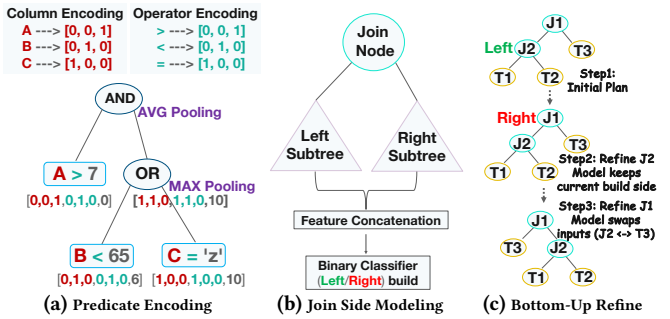


Figure 4: Intelligent Optimization by History Executions.

5.2 Intelligent Optimization

To enhance the robustness and accuracy of query optimization in dynamic analytical workloads, ByteHouse adopts a history-based optimization (HBO) [21, 55] that leverages runtime statistics collected from past query executions to guide future cost-based decisions. Each plan fragment is transformed into a canonical representation and assigned a hash value that serves as a lookup key in a lightweight history store. During query compilation, the optimizer performs hash-based matching to retrieve previously recorded statistics, such as predicate selectivities and join cardinalities, and incorporates them into the cost estimation. While HBO improves the robustness and accuracy of cost estimation by reusing statistics from past executions, it remains fundamentally limited to plan fragments observed before and cannot fully accommodate new query patterns or evolving data distributions.

To overcome this limitation, ByteHouse incorporates regression models to learn correlations among query structures, data characteristics, and runtime behavior [41, 42, 49, 60, 64, 78, 79]. By capturing these learned relationships, ByteHouse can generalize beyond hash-based plan reuse and provide more accurate cost estimates for previously unseen workloads. Beyond ByteCard [27], which leverages probabilistic graph models [32] to refine cardinality estimation for base tables (e.g., filter selectivity and join size), we further deploy learning-based optimizations targeting Predicate Pushdown Selection (PPS) and Join Side Selection (JSS). Both optimizations have been validated in production deployments.

1) Predicate Pushdown Selection. While predicate pushdown [29, 71] is commonly regarded as a straightforward optimization, its impact in ByteHouse is more nuanced due to disaggregated storage and multimodal execution. In particular, aggressively pushing down predicates whose evaluation cost is high relative to their

selectivity benefit (e.g., vector similarity conditions or UDF-based filters) may increase remote I/O and processing overhead while precluding more effective pruning opportunities later in the plan. Therefore, predicate pushdown in ByteHouse requires cost-aware selection rather than indiscriminate application.

We formulate this decision process as a supervised regression task for estimating predicate-specific I/O costs. In the offline stage, we collect training samples of (predicate, I/O cost) pairs, and train a regression model that learns the mapping from predicate features to observed scan costs. At runtime, ByteHouse leverages the trained model to evaluate candidate predicates and select the most cost-effective ones for pushdown.

To construct candidate predicates, ByteHouse decomposes each query's WHERE clause into top-level conjunctive components separated by AND operators, yielding independent candidates for pushdown evaluation. Each candidate is represented as an Abstract Syntax Tree (AST) whose nodes correspond to comparison operators, column references, literal values, etc. For node encoding, we apply one-hot encodings to categorical features such as comparison operators and column identifiers, while continuous features are discretized into value-domain buckets following prior work [31, 70]. For AST encoding, conjunctions and disjunctions (AND, OR) are modeled as pooling functions [23] over their child encodings rather than as categorical tokens. This approach can preserve the logical semantics of predicate composition. Specifically, OR applies MAX pooling and AND applies AVG pooling. The final predicate embedding is obtained by a postorder traversal across the AST that recursively aggregates node representations. Figure 4a presents an example of predicate encoding for: (A > 7) AND (B < 65 OR C='x'). Note, the underlying relational table contains three columns (A, B, C), and the analytical workload involves three comparison operators (>, <, =); columns A and B share a numeric domain of $[0, 100]$, and C is a categorical attribute with 10 distinct values.

2) Join Side Selection. Join performance is highly sensitive to which input is chosen as the *build* (hashed) side versus the probe side, as this decision governs hash-table size, memory pressure, and downstream I/O. Traditional optimizers rely on cost models to select the build side, but their effectiveness hinges on accurate cardinality estimates and may degrade under skew or complex predicates. In ByteHouse, we introduce a learning-based side-selection mechanism that refines the initial join physical plan produced by the optimizer. As illustrated in Fig. 4b, we formulate build/probe selection for a join node as a binary classification task that predicts which child should serve as the build input. For each join node, the model constructs a feature vector by concatenating the learned representations of its left and right subtrees with features of the current join (e.g., join predicates, estimated selectivities, and row-width signals) in post-order; this embedding is then fed to a classifier that outputs *left-build* or *right-build*.

During training, labels are derived offline by comparing the observed output cardinalities of the two subtrees: if the left subtree's cardinality is smaller, the label is *left-build*; otherwise, *right-build*. To maintain semantic consistency across recursive subplans, we assume that when evaluating a join node, all its descendant joins have already been assigned correct side selections. This assumption is enforced during training by restructuring query plans to satisfy it, and during inference by traversing the plan bottom-up (see Fig. 4c), ensuring that lower-level joins are decided before higher-level ones. In this manner, the model adaptively determines build/probe selection throughout the query plan, improving memory efficiency and hash-table locality while preserving overall query semantics.

6 Multi-Modal Data Indexing and Hybrid Search

In ByteHouse's production scenarios, there is a growing need to optimize the execution of multimodal search queries (e.g., text+vector retrieval in code assistant). While existing vector databases (e.g., Milvus-based solutions) are primarily designed for standalone similarity search, they lack tight integration with relational query execution. In contrast, ByteHouse builds upon a document-chunk abstraction (see Section 3.1.1) and extends standard SQL to support both vector and full-text search primitives within a single query. To efficiently execute such queries, ByteHouse introduces modality-aware optimizations, including runtime filtering, hybrid vector index designs, and multi-source result fusion strategies.

In the following, we introduce the design choices of vector indexes (in a hierarchical way) and hybrid data search in ByteHouse.

Multi-Layer Vector Indexes. Within a single table, we determine the suitable index structures based on the service types. The candidate vectors undergo a multi-layer structure tailored for various latency, freshness, and cost trade-offs. First, a coarse index layer performs partition-level pruning using product quantization (PQ) and BLAS-accelerated centroid distance, avoiding unnecessary index traversal. The rest layers of the index are determined based on the service types: (*i*) For latency-critical online services that demand high recall and millisecond-level responsiveness, ByteHouse adopts HNSW with scalar quantization (SQ), where vectors are pre-quantized into compact representations and organized in a navigable small-world graph. This allows bounded-depth traversal during query time, achieving fast search while maintaining recall. Under this setting, we can build the index asynchronously and decouple it from ingestion, ensuring minimal impact on write throughput. (*ii*) For near real-time services requiring high recall and rapid vector visibility (within seconds to sub-seconds), but with relaxed latency (e.g., 100ms-1s), ByteHouse adopts centroid-based partitioning during index construction, with each partition maintaining either full-precision vectors (IVFFlat), quantized vectors (IVFSQ), or PQ-based compression (IVFPQ). This enables fast ingestion-to-query cycles and moderate memory usage. (*iii*) For cost-sensitive services, ByteHouse adopts DiskANN that stores full-precision graph structures on SSDs and caches routing metadata in memory. Additionally, we use optimized I/O prefetching for common queries and beam search to maintain latency under relaxed constraints. For even more relaxed workloads (e.g., over long-tail vectors order than six months), ByteHouse supports DiskIVFSQ to store quantized, partitioned vectors on disk, significantly reducing both memory and compute costs, making it suitable for archival data with minimal freshness and latency demands.

Upon carefully constructed vector indexes, ByteHouse further extends its support to more complicated hybrid data search queries that blend semantic, textual, and structured constraints. Consider the hybrid query shown in Figure 5. The nested subquery retrieves

```sql
SELECT *
FROM (
    SELECT a.document_id, a.chunk_id, a.chunk_content
    FROM document_table
    ORDER BY RANK_FUSION(
           cosineDistance(a.embedding_col, [0.114, 0.257, ...]) ASC  WEIGHT 1.0,
           textSearch(a.text_col, 'ByteDance')              DESC WEIGHT 2.0
       ) DESC
    LIMIT 100
) AS a
JOIN (
    SELECT *
    FROM label_table
    WHERE label_value = 'doc_image'
) AS b
ON a.document_id = b.document_id
   AND a.chunk_id = b.chunk_id;
```

Figure 5: Example Hybrid Data Search in ByteHouse.

the top-K chunks based on a fused ranking over both semantic similarity (via vector-based cosine distance) and keyword relevance (via lexical text search). These two modalities are integrated through the RANK_FUSION operator, which combines scores with configurable weights to balance semantic and lexical contributions. For the outer query, instead of issuing this vector-augmented query in isolation, we join its results with a filtered label table that selects only chunks annotated with a specific tag (e.g., "doc_image"). This join is structured to enable scalar-predicate-aware optimizations. ByteHouse executes such hybrid queries in three main steps.

(1) Cross-Table Runtime Filtering. First, ByteHouse conducts cross-table runtime filtering to prune the search space before initiating vector retrieval. The optimizer estimates the selectivity and optimal join order [46] (see Section 5). When the scalar-side table (e.g., label_table) is more selective, a runtime filter (e.g., Bloom filter, bitmap) is generated over the join keys and injected into the scan operator of the document_table. This early-stage pruning can both (1) improve the accuracy and (2) eliminate irrelevant rows. Otherwise, ByteHouse injects runtime filtering directly into operators like vector index scan, enabling coarse-grained pruning even within the vector retrieval phase.

(2) Fusion-based Hybrid Data Search. With the RANK_FUSION operator (used in Figure 5), ByteHouse brings vector and text search together with relation query processing. Specifically, RANK_FUSION is a specialized variant of the relational Union operator that offers two alternative fusion strategies. The first is a score-based fusion approach, which performs Min–Max normalization within each modality to align score scales, followed by weighted linear aggregation of the normalized scores. Alternatively, ByteHouse employs a rank-based Reciprocal Rank Fusion (RRF) algorithm [17], which aggregates results purely based on their relative positions in ranked lists. Given n ranked lists (e.g., from text and vector indexes), RRF assigns each document d a unified relevance score: $RRF(d) = \sum_{i=1}^{n} \frac{1}{k+r_i(d)}$, where $r_i(d)$ denotes the rank position of document d in the i-th modality-specific retrieval list, k is a smoothing constant (typically set to 60) that controls the decay of lower-ranked items, and n is the number of participating modalities. By operating on rank positions rather than absolute similarity scores, RRF achieves modality-agnostic and calibration-free fusion.

(3) Selective Post-Join Refinement. After the hybrid retriever produces the top-K candidates, ByteHouse performs a selective post-join refinement to enforce structured predicates. Because runtime filtering (Step (1)) has already pruned most irrelevant rows, this join operates on a substantially reduced candidate set.

7 Performance Study

7.1 Standard Benchmark Comparison

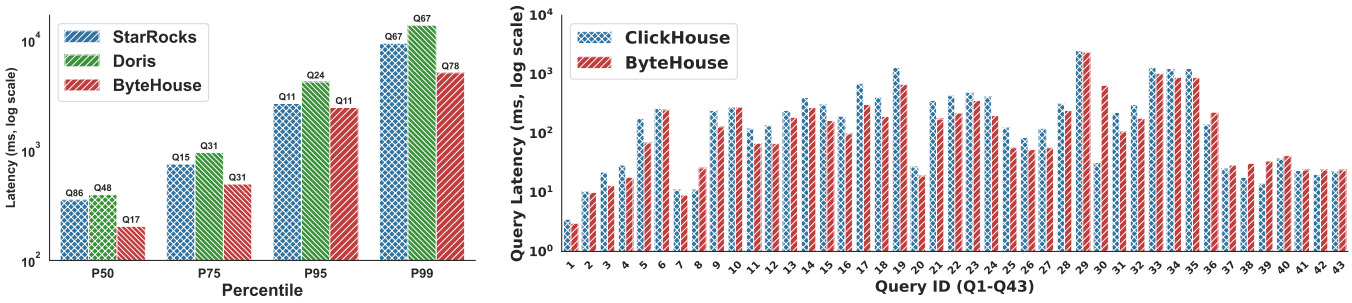
We evaluate the analytical performance of ByteHouse operating against the latest StarRocks [59], Doris [7], and ClickHouse [52] using two representative benchmarks: TPC-DS (SF-1000) [63] and ClickBench [61]. All experiments are conducted on a dedicated cluster equipped with a 64-core AMD EPYC 9Y24 processor (128 hardware threads at 2.6 GHz) and 247 GB of main memory. Figure 6 summarizes the comparative results. On TPC-DS SF-1000 (Fig. 6a), ByteHouse consistently achieves lower end-to-end latency across all reported percentiles. The performance gaps widen at higher percentiles, such as P95 and P99, where multi-stage aggregations and joins dominate execution time and amplify engine-level bottlenecks. In these scenarios, ByteHouse achieves substantially lower tail latency than both StarRocks and Doris, indicating stable performance even under the most demanding queries. Fig. 6b presents per-query results on the ClickBench benchmark, which consists of 43 queries executed five times each, with the fastest run recorded. ByteHouse outperforms ClickHouse on approximately 32 queries and reduces the overall end-to-end latency by 25.4%. These performance gains are driven by ByteHouse's high-parallelism execution engine, adaptive aggregation and grouping mechanisms, optimized Top-N processing paths, and an efficient memory allocator that reduces allocation overhead and fragmentation under high concurrency.

7.2 Effect of Incremental Processing

Figure 7 evaluates the efficiency of incremental processing mode (IPM) under representative inner join workloads from the TPC-H benchmark at scale factor 100. Figure 7a compares the CPU time of full recomputation and incremental computation across three typical join queries (Q12, Q14, Q19), where updates are applied to the lineitem table with an update ratio of 2.5%. The incremental approach consistently outperforms the full recomputation baseline, reducing CPU time by 28.4%–69.2%, which highlights its advantage for processing complex multi-join plans. Figure 7b further analyzes how incremental processing scales with different update ratios (2.5%, 5%, 10%) using join-only SQL queries without any filter predicates, with updates applied to either orders or lineitem. We measure the CPU time when applying updates to either the orders or lineitem table in query Q12. As the update ratio increases from 2.5% to 10%, the CPU time reduction grows from 19.9% up to 61.7%, indicating that the incremental strategy becomes increasingly beneficial as the magnitude of data updates rises. These results collectively demonstrate that incremental processing substantially improves computational efficiency, particularly in update-intensive analytical workloads.

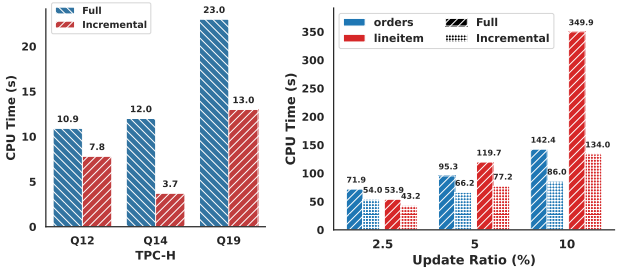
7.3 Effect of CrossCache

We evaluate the performance of CrossCache on a dedicated ByteHouse cluster used for large-scale advertising analytics, including user conversion analysis across ad impressions, clicks, and exposure



(a) TPC-DS Benchmark (SF-1000). (b) ClickBench Benchmark (Overall 25.4% Latency Reduction).

Figure 6: Comparative Results on Standard Benchmarks.



(a) Inner Join Performance. (b) Impact of Update Ratio.

Figure 7: Efficiency of Incremental Processing.

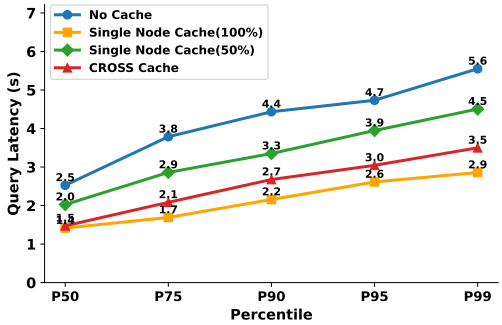
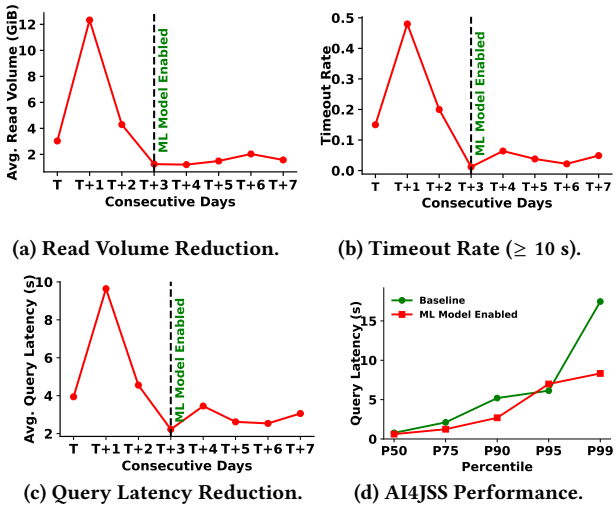


Figure 8: Latency Evaluation of CrossCache.

events. The cluster consists of 180 compute nodes, each equipped with 60 CPU cores (AMD EPYC 9Y24 96-Core Processor, 3.69 GHz), 256 GB of memory, and a 2.5 TB SSD. CrossCache is deployed with one Cache Coordinator and 47 Cache Nodes, where each node provides 2.25 TB of local SSD cache. The evaluated workload comprises the top 1000 historical queries with the largest data scan volumes. Each query completes within 5–7 s when served from cache, while full execution without caching typically takes around one minute. To isolate the effect of distributed caching, we evaluate CrossCache and the single-node cache baselines in a fully isolated environment and enable only one caching mode per run to avoid cross-interference.

Figure 8 reports end-to-end query latency percentiles (P50–P99) across four mutually exclusive settings: no cache, a single-node cache with a 100% hit ratio, the same single-node cache with a 50% hit ratio, and CrossCache. The cache hit ratio is controlled by adjusting the available cache capacity. Across all configurations, caching substantially reduces query latency. Compared with the single-node cache at a 50% hit ratio, CrossCache reduces latency across all percentiles, with improvements of about 25% at P50, 18% at P90, and 22% at P99. While CrossCache exhibits marginally higher latency than the idealized single-node cache with a perfect (100%) hit ratio, it consistently achieves lower latency than the single-node cache at a 50% hit ratio. This advantage is particularly meaningful in production environments, where maintaining perfect cache locality on a single node is rarely achievable.

7.4 Effect of AI-driven Optimization



(a) Read Volume Reduction. (b) Timeout Rate (≥ 10 s).

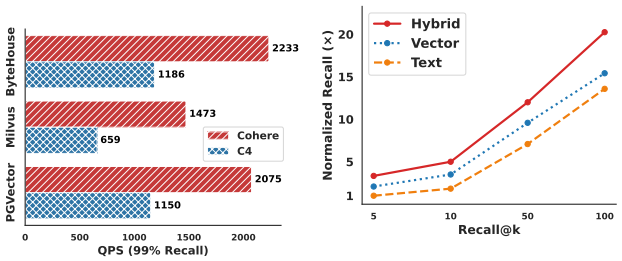
(c) Query Latency Reduction. (d) AI4JSS Performance.

Figure 9: Efficiency of AI-driven Optimization.

Figure 9 illustrates the performance of two AI-driven approaches on query execution efficiency. The experiment was conducted on a production cluster consisting of 100 nodes, each equipped with 192 CPU cores and 1.48 TiB of memory. Figures 9a-9c present the results of the Predicate Pushdown Selectivity (PPS) experiment, where an ML model predicts filter selectivity and dynamically adjusts predicate pushdown strategies. After enabling the model (day T+3), the system exhibits notable reductions in both I/O and query latency: the average read volume drops by up to 87%, the proportion of queries exceeding 10 seconds (timeout rate) drops sharply, and the average query latency is reduced accordingly. Figure 9d further evaluates the Join Side Selection (JSS) experiment, where an ML model is employed to optimize build/probe side selection.

The experiment evaluates 1,000 join queries sampled from real production traces, comparing the latency distributions of the proposed approach against the baseline optimizer. Across all percentiles, the AI-driven approach achieves 15%–45% lower query latency, with particularly strong improvements at the tail (P95–P99) where long-running queries dominate overall cost. Across all latency percentiles (P50–P99), the ML model consistently outperforms the baseline, with the most significant gains observed in the tail latencies (P95–P99). Overall, these results confirm that AI-driven optimization substantially improves scan and join efficiency, delivering measurable gains in real-world analytical query processing.

7.5 Hybrid Multimodal Query Processing



(a) Vector Retrieval Throughput. **(b) Hybrid Search Recall.**

Figure 10: Performance of Multimodal Query Processing.

Figure 10 evaluates the multimodal query processing capability of ByteHouse. Figure 10a reports vector retrieval throughput (QPS at 99 % recall) on the Cohere [15] and C4 [5] datasets using the VDBBench framework [80], comparing ByteHouse with Milvus [67] and pgvector [47]. The workload consists of hybrid queries that combine vector similarity search with a 1% scalar filter. Across both datasets, ByteHouse achieves the highest throughput, improving performance by 50–60% over Milvus on Cohere and by more than 50% on C4, while also maintaining a clear lead over pgvector. These gains stem from ByteHouse's multi-layer vector index choices and cross-table runtime filtering.

Figure 10b presents the multimodal retrieval accuracy using the MS MARCO [8], which contains 8.84M passages (we sample 2M for testing). Each passage is represented by the BGE-M3 embedding model [13]. We evaluate three retrieval approaches in ByteHouse (i.e., Vector Search, Text search, and Hybrid Search) under the recall metric at different top-k thresholds. The results shows that Hybrid Search consistently achieves the best overall recall, outperforming the single-modality baselines. At Recall@100, it yields approximately a 30% improvement over vector search and a 50% improvement over text search. These results demonstrate that ByteHouse effectively integrates semantic embedding signals with lexical matching within a unified execution framework.

8 Lessons Learned

Durability and Data Organization. Our experience indicates that storage latency is dominated by the write path, where durability and isolation guarantees must be enforced during ingestion. Write-through policies [16, 65] simplify reliability but often suffer from high cold-read latency under cache misses, whereas write-back policies [6, 45] hide write latency through buffering and deferred persistence, which increases the complexity of durability semantics. A key lesson across these policies is that durability and availability should be decoupled so that persistence can proceed asynchronously without compromising service continuity. We also believe that hybrid data layouts [50] that keep hot tuples in a row format and colder regions in a compressed columnar format provide a practical way to accommodate both update-intensive operations and bandwidth-efficient analytical scans.

Sustaining High Concurrency. ByteHouse serves analytical workloads that reach $10^4 - 10^5$ read/write QPS, where per-query control overheads, metadata lock contention, and I/O amplification from small fragmented accesses quickly become bottlenecks. Our experience indicates that achieving stable performance at this scale requires coordinated design across planning, compute, and storage, rather than isolated operator-level optimizations. ByteHouse reduces planning-layer control overheads through *prepared statements*, accelerates execution-layer performance for selective workloads (e.g., TopN) via *short-circuit evaluation*, and employs the CrossCache layer to absorb fine-grained accesses and reduce storage-layer I/O amplification. This end-to-end co-design enables scalable throughput with stable tail latency at high concurrency.

Vector Search and Multi-Layer Indexing. A key lesson is that vector computation is becoming a fundamental building block for multimodal analytics. Similarity search workloads are moving beyond simple Top-k retrieval with scalar filters toward richer hybrid patterns, such as similarity-bounded range aggregation [35, 39], vector-driven word-cloud analysis [12, 51, 69], and multi-vector joint recall across embedding spaces [14, 38]. These trends highlight the need for more flexible vector-query semantics and tighter integration of vector search with traditional analytical operators to support increasingly expressive multimodal workloads.

9 Conclusions

This paper presents ByteHouse, a cloud-native shared-storage data warehouse for real-time multimodal analytics. ByteHouse integrates a vertically optimized storage layer with a unified execution framework that supports analytic, batch, and incremental processing. Its fusion-based retrieval operators and hybrid optimization techniques enable efficient multimodal query execution. Evaluations across production and standard benchmarks confirm that ByteHouse is a scalable, high-performance foundation for emerging intelligent data applications.

Acknowledgments

Xuanhe Zhou is the corresponding author. We thank Tao Wang and Xuechao Lian for contributions to the development of SBM and IPM, Lixiang Qian for APM, Chengxian Li for traditional optimizer, Changheng Cai for hybrid data search, Zichun Huang for NexusFS, and Xuewu Xiao, Qiang Ren, and Shuo Jia for CrossCache. This work was supported in part by National Key R&D Program of China (No. 2023YFB4502400), Shanghai Jiao Tong University AI for Engineering Initiative and Shanghai Qi Zhi Institute Innovation Program.

References

[1] 2025. Amazon Redshift. https://aws.amazon.com/redshift/. Accessed: 2025-10-27.

[2] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. 2023. ALP: Adaptive lossless floating-point compression. *SIGMOD* 1, 4 (2023), 1–26.

[3] 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.

[4] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, et al. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[5] Allen Institute for AI. 2020. C4 Dataset: Colossal Clean Crawled Corpus. https://huggingface.co/datasets/allenai/c4. Accessed: 2025-10-27.

[6] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[7] Apache Doris Community. 2017. Apache Doris: An MPP Analytical Database for Real-Time Analytics. https://github.com/apache/doris. Accessed: 2025-10-27.

[8] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. 2016. MS MARCO: A human generated machine reading comprehension dataset. *arXiv preprint arXiv:1611.09268* (2016).

[9] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[10] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[11] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614.

[12] Quim Castella and Charles Sutton. 2014. Word storms: Multiples of word clouds for visual comparison of documents. In *Proceedings of the 23rd international conference on World wide web*. 665–676.

[13] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. arXiv:2402.03216 [cs.CL]

[14] Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics ACL 2024*. 2318–2335.

[15] Cohere Team. 2022. Wikipedia Dataset (December 2022). https://huggingface.co/datasets/Cohere/wikipedia-22-12. Accessed: 2025-10-27.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[17] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. 758–759.

[18] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[19] Amr El-Helw, Venkatesh Raghavan, Mohamed A Soliman, George Caragea, Zhongxian Gu, and Michalis Petropoulos. 2015. Optimization of common table expressions in mpp database systems. *VLDB* 8, 12 (2015), 1704–1715.

[20] Pit Fender and Guido Moerkotte. 2011. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). 864–875.

[21] César A Galindo-Legaria, Milind M Joshi, Florian Waas, and Ming-Chuan Wu. 2003. Statistics on views. In *Proceedings 2003 VLDB Conference*. Elsevier, 952–962.

[22] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370* (2025).

[23] Hossein Gholamalinezhad and Hossein Khosravi. 2020. Pooling methods in deep neural networks, a review. *arXiv preprint arXiv:2009.07485* (2020).

[24] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[25] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[26] Anja Gruenheid, Jesús Camacho-Rodríguez, Carlo Curino, Raghu Ramakrishnan, Stanislav Pak, Sumedh Sakdeo, Lenisha Gandhi, Sandeep K Singhal, Pooja Nilangekar, and Daniel J Abadi. 2025. AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes. In *Companion of the 2025 International Conference on Management of Data*. 404–417.

[27] Yuxing Han, Haoyu Wang, Lixiang Chen, Yifeng Dong, Xing Chen, Benquan Yu, Chengcheng Yang, and Weining Qian. 2024. ByteCard: Enhancing ByteDance's Data Warehouse with Learned Cardinality Estimation. In *Companion of the 2024 International Conference on Management of Data*. 41–54.

[28] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskAnn: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[29] Ipoom Jeong, Jinghan Huang, Chuxuan Hu, Dohyun Park, Jaeyoung Kang, Nam Sung Kim, and Yongjoo Park. 2025. UPP: Universal Predicate Pushdown to Smart Storage. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 419–433.

[30] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. 2020. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. VLDB Endow.* 13, 12 (2020), 3272–3284.

[31] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*.

[32] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[33] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. Btrblocks: Efficient columnar compression for data lakes. *SIGMOD* 1, 2 (2023), 1–26.

[34] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801. doi:10.14778/2367502.2367518

[35] Hai Lan, Shixun Huang, Zhifeng Bao, and Renata Borovica-Gajic. 2024. Cardinality Estimation for Similarity Search on High-Dimensional Data Objects: The Impact of Reference Objects. *Proceedings of the VLDB Endowment* 18, 3 (2024), 544–556.

[36] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[37] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proc. VLDB Endow.* 17, 12 (2024), 3786–3798.

[38] Minghan Li, Sheng-Chieh Lin, Xueguang Ma, and Jimmy Lin. 2023. Slim: Sparsified late interaction for multi-vector retrieval with inverted indexes. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1954–1959.

[39] Anqi Liang, Pengcheng Zhang, Bin Yao, Zhongpu Chen, Yitong Song, and Guangxu Cheng. 2024. UNIFY: Unified Index for Range Filtered Approximate Nearest Neighbors Search. *arXiv preprint arXiv:2412.02448* (2024).

[40] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[41] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718.

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[43] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[44] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. 2020. Dremel: A decade of interactive SQL analysis at web scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472.

[45] Matteo Merli, Sijie Guo, Penghui Li, Hang Chen, and Neng Lu. 2025. Ursa: A Lakehouse-Native Data Streaming Engine for Kafka. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5184–5196.

[46] Zhaojie Niu, Xinhui Tian, Xindong Peng, and Xing Chen. 2025. BlendHouse: A Cloud-Native Vector Database System in ByteHouse. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 4332–4345.

[47] pgvector Team. 2023. pgvector: Open-source vector similarity search for PostgreSQL. https://github.com/pgvector/pgvector. Accessed: 2025-10-27.

[48] Martin Prammer, Xinyu Zeng, Ruijun Meng, Wes McKinney, Huanchen Zhang, Andrew Pavlo, and Jignesh M Patel. 2025. Towards Functional Decomposition of Storage Formats. In *Conference on Innovative Data Systems Research (CIDR)*.

[49] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data*. 225–237.

[50] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3290–3303.

[51] Erich Schubert, Andreas Spitz, Michael Weiler, Johanna Geiß, and Michael Gertz. 2017. Semantic word clouds with background corpus normalization and t-distributed stochastic neighbor embedding. *arXiv preprint arXiv:1708.03569* (2017).

[52] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. Clickhouse-lightning fast analytics for everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[53] Damien Sereni, Pavel Avgustinov, and Oege De Moor. 2008. Adding magic to an optimising datalog compiler. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 553–566.

[54] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*. 435–446.

[55] Pranjal Shankhdhar, Feilong Liu, Jay Narale, James Sun, Rebecca Schlussel, and Lyublena Antova. 2024. Presto's History-Based Query Optimizer. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4077–4089.

[56] Yasin N Silva, Paul-Ake Larson, and Jingren Zhou. 2012. Exploiting common subexpressions for cloud query processing. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1337–1348.

[57] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 337–348.

[58] Daniel Sotolongo, Daniel Mills, Tyler Akidau, Anirudh Santhiar, Attila-Péter Tóth, Botong Huang, Boyuan Zhang, Igor Belianski, Ling Geng, Matt Uhlar, et al. 2025. Streaming Democratized: Ease Across the Latency Spectrum with Delayed View Semantics and Snowflake Dynamic Tables. In *Companion of the 2025 International Conference on Management of Data*. 622–634.

[59] StarRocks Team. 2020. StarRocks: A High-Performance MPP Database for Analytics. https://github.com/StarRocks/starrocks. Accessed: 2025-10-27.

[60] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[61] ClickHouse Team. 2021. ClickBench: A Benchmark for Analytical DBMS. https://github.com/ClickHouse/ClickBench. Accessed: 2025-10-27.

[62] The Apache Software Foundation. 2025. Apache Arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/. Accessed: 2025-10-24.

[63] Transaction Processing Performance Council. 2018. TPC-DS Benchmark (Version 3.2). http://www.tpc.org/tpcds/. Accessed: 2025-10-27.

[64] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1153–1170.

[65] 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.

[66] Vortex Project Team. 2024. Vortex: A Next-Generation High-Performance Data Format for AI and Analytics. https://vortex.dev.

[67] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 international conference on management of data*. 2614–2627.

[68] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, et al. 2020. Tempura: a general cost-based optimizer framework for incremental data processing. *Proceedings of the VLDB Endowment* 14, 1 (2020), 14–27.

[69] Yingcai Wu, Thomas Provan, Furu Wei, Shixia Liu, and Kwan-Liu Ma. 2011. Semantic-preserving word clouds by seam carving. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 741–750.

[70] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743* (2020).

[71] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate pushdown for data science pipelines. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[72] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[73] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data*. 295–308.

[74] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2026. F3: The Open-Source Data File Format for the Future. (2026). SIGMOD).

[75] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.

[76] Yiwen Zhang, Guokuan Li, Kai Lu, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. 2024. PhatKV: Towards an Efficient Metadata Engine for KV-based File Systems on Modern SSD. In *Proceedings of the MSST 2024*. IEEE.

[77] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 1060–1071.

[78] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.

[79] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *VLDB* 14, 9 (2021), 1489–1502.

[80] Zilliz. 2023. VectorDBBench: A Benchmark Suite for Vector Databases. https://github.com/zilliztech/VectorDBBench. Accessed: 2025-11-17.