OmniTable: A Unified Wide-Table System for Petabyte-Scale LLM Data Curation and Exploration

Yuzhuo Fu
fuyuzhuo.fyz@antgroup.com
AntGroup

Xiangchun Wang
wangxiangchun.wxc@antgroup.com
AntGroup

Chao Huang
zhonggong.hc@antgroup.com
AntGroup

Liyi Wang
banzhu.wly@antgroup.com
AntGroup

Binwei Zeng
binwei.zbw@antgroup.com
AntGroup

Yuhan Wang
heqi.wyh@antgroup.com
AntGroup

Taotao Nie
nietaotao.ntt@antgroup.com
AntGroup

Dongke Hu
dongke.hudk@antgroup.com
AntGroup

Wang Hong
hongwang.hw@antgroup.com
AntGroup

Jiayi Wang
jiaer.wjy@antgroup.com
AntGroup

Wenwen Cui
cww516675@antgroup.com
AntGroup

Zhuyan Zhou
zhouzhuyan.zzy@antgroup.com
AntGroup

Yushun Guo
guoyushun.gys@antgroup.com
AntGroup

Yuhan Xing
xingyuhan.xyh@antgroup.com
AntGroup

Jiaxin Lian
lianjiaxin.ljx@antgroup.com
AntGroup

Peng Lin
peng.linlp@antgroup.com
AntGroup

Qing Cui
cuiqing.cq@antgroup.com
AntGroup

Wenhui Shi
yushun.swh@antgroup.com
AntGroup

ABSTRACT

Data curation is a critical bottleneck in industrial-grade LLM development, where petabyte-scale unstructured corpora are scattered across hundreds of physical tables, feature engineering relies on manual, table-centric pipeline orchestration, and data lineage is largely absent. We present OmniTable as an architecture blueprint for a unified wide-table layer built on Logical Unification, Physical Separation, targeting petabyte-scale LLM data curation and exploration. OmniTable makes four contributions: (1) a unified wide-table abstraction that consolidates multi-source heterogeneous data and thousands of derived features under a single logical schema via logical-physical mapping; (2) declarative feature lifecycle management that automates dependency resolution, execution planning, operator fusion, and lineage tracking, replacing manual pipeline orchestration with a "declare-and-execute" paradigm; (3) an adaptive execution engine with autonomous governance that achieves stable PB-scale feature backfill through heterogeneous compute routing (CPU/GPU), adaptive tuning, UDF-level fault tolerance, and automated storage layout optimization; and (4) hybrid-accelerated data exploration combining a global ID index, transparent OLAP offloading, and background materialized views to deliver second-level point lookups and filtered exports exceeding 20 TB/hour. In production, OmniTable manages over 35 PB of training data across web, code, PDF, and SFT domains, reducing the human-in-the-loop curation cycle from approximately 14 days to approximately 2.5 days (5.6x over the pre-OmniTable production workflow), with consistent feature versioning, auditable lineage, and minimal manual intervention.

PVLDB Reference Format:
Yuzhuo Fu, Xiangchun Wang, Chao Huang, Liyi Wang, Binwei Zeng, Yuhan Wang, Taotao Nie, Dongke Hu, Wang Hong, Jiayi Wang, Wenwen Cui, Zhuyan Zhou, Yushun Guo, Yuhan Xing, Jiaxin Lian, Peng Lin, Qing Cui, and Wenhui Shi. OmniTable: A Unified Wide-Table System for Petabyte-Scale LLM Data Curation and Exploration. PVLDB, 19(12): 4276 - 4289, 2026.
doi:10.14778/3827998.3828032

1 INTRODUCTION

Large Language Models (LLMs) such as GPT-4 [2] and Llama-3 [31] follow scaling laws [35, 38]: their capabilities improve with more parameters, compute, and training data [17, 45]. As architectures converge [66], *data curation*—collecting, cleaning, enriching, and filtering heterogeneous corpora—has become a key bottleneck [43, 49, 51, 69]. At petabyte (PB) scale, structured-data ETL

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828032

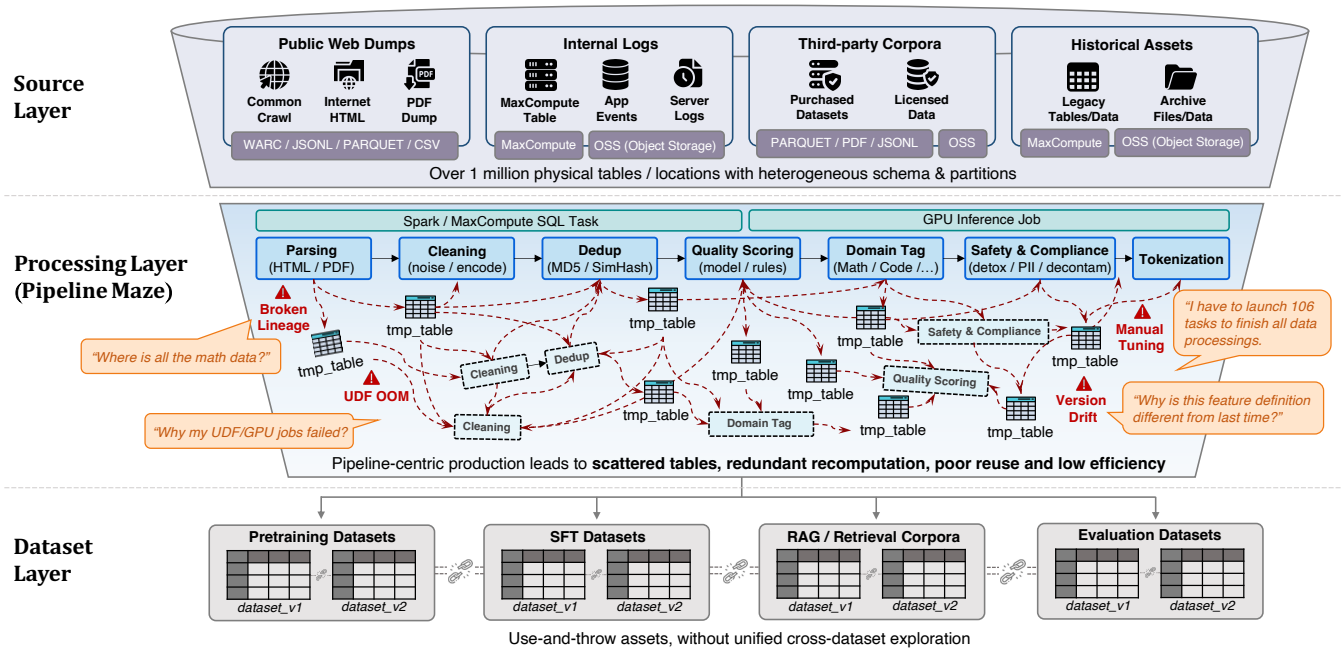

Figure 1: The Pipeline Maze in industrial LLM data preprocessing. Data from heterogeneous sources (top) flows through fragmented processing pipelines (middle), producing isolated, disposable datasets (bottom).

paradigms are inadequate for LLM governance [72]. Existing lakehouse formats (e.g., Iceberg/Delta) provide ACID and schema evolution but are largely *table-centric*. In LLM curation, iteration is *feature*-centric: columns are computed via UDF dependency DAGs and often require CPU/GPU routing. OmniTable targets these gaps with Catalog-driven logical unification and feature-centric execution.

In practice, industrial LLM data preparation workflows often degrade into what we term a *pipeline maze*, characterized by three pain points. **(1) Data silos**: corpora from dozens of sources are scattered across hundreds of physical tables, making cross-dataset discovery extremely difficult. **(2) Costly feature engineering**: adding a single new feature requires manually coordinating tasks across every relevant dataset—in one case, an engineer had to "drag and drop 106 tables onto the task canvas" for a single feature. **(3) Broken lineage**: UDF logic is spread across codebases without centralized version control, so feature definitions become inconsistent, and data lineage and feature definitions are not systematically captured; as a result, teams cannot reliably trace how each data/feature iteration affects downstream training runs and model quality [54, 56, 65].

This paper makes the following contributions:

- **Unified Wide-Table Abstraction.** A logical data model that consolidates multi-source raw data, multi-stage processing results, and thousands of derived features under a single schema, eliminating data silos (§4.1–4.2).
- **Feature Lifecycle Management.** A feature-centric paradigm in which users declare computation logic and the system automatically resolves dependencies, generates execution plans, performs operator fusion, and maintains full lineage (§4.3).
- **Autonomous Governance and Adaptive Execution Engine.** An engine that intelligently routes tasks across CPU/GPU backends with adaptive tuning and UDF-level fault tolerance, coupled with autonomous background storage optimization for long-term PB-scale scalability (§4.5).
- **High-Performance Hybrid-Accelerated Exploration.** A query service combining global ID indexing, transparent OLAP offloading, and background materialization to deliver second-level point lookups and filtered exports exceeding 20 TB/hour (§4.4).

The rest of this paper is organized as follows. §2 motivates the design through real-world challenges. §3 presents the system overview. §4 details the design and implementation. §5 reports experimental results. §6 shares lessons learned. §7 discusses related work, and §8 concludes.

2 MOTIVATION AND CHALLENGES

2.1 Workflow of LLM Data Preprocessing

In industrial LLM development, preprocessing resembles a funnel [14, 50, 64]: raw corpora undergo parsing, cleaning, deduplication [16, 20, 40], scoring, tokenization [59, 60, 74], and sample assembly. In practice it becomes a fragmented *pipeline maze* without a unified abstraction, where each iteration produces disposable tables and ad-hoc jobs.

Figure 1 summarizes the flow from heterogeneous sources [5, 39, 44] through multiple engines (Spark [73], MaxCompute SQL [68], GPU inference) to isolated outputs. This leads to hard discovery, per-table feature backfill (e.g., adding PPL [22, 42]), and difficult fault localization due to missing lineage [51, 56].

2.2 Challenges

We distill the above pain points into four challenges that define OmniTable's design requirements.

C1: Heterogeneity and fragmentation. LLM corpora come from diverse sources (public crawls, internal logs, procured data) in heterogeneous storage formats and are processed on both CPU and GPU engines with inconsistent runtime environments and failure semantics [37, 43]. Without a unified data abstraction, each source or engine requires a separate pipeline, preventing users from obtaining a consistent view across sources and processing stages.

C2: Scale and performance. The system must handle hundreds of PB with continuous writes at PB/day scale and thousands of logical columns [25, 63], while simultaneously supporting interactive exploration (second-level point lookups) and high-throughput export (≥20 TB/h). Frequent incremental writes introduce small-file and partition explosion, requiring scalable performance under mixed workloads.

C3: Agility and iteration speed. Researchers need to frequently validate different data subsets, filtering strategies, and feature sets through ablation experiments, ideally on a daily basis [53]. In traditional workflows, however, the unit of iteration is individual tables and tasks, so adding or modifying a feature requires re-orchestrating pipelines across all affected datasets, with cost scaling linearly or super-linearly with the number of datasets. The system should shift the iteration unit from physical tables to declarative feature operations.

C4: Manageability and traceability. Feature logic is distributed across codebases without centralized version control. Stale lineage [54, 65] causes definition drift and irreproducible results. The full feature lifecycle—from declaration through computation to backfill—must automatically generate queryable lineage records for troubleshooting, auditing, and reproduction.

3 SYSTEM OVERVIEW

To address the challenges identified in §2, OmniTable adopts a unified wide table as its core abstraction and, crucially, makes the system *feature-centric*: columns are first-class assets with registered UDF definitions, dependency DAGs, versioning, and engine preferences (CPU/GPU). This shifts the lifecycle from pipeline-centric "tables and tasks" to continuous evolution of data assets (rows) and feature assets (columns).

3.1 Logical Unification, Physical Separation

OmniTable presents a single logical wide-table view while allowing diverse physical layouts underneath. Physical separation and query-driven materialization may add ~8–15% storage for hot column groups, but reduce end-to-end curation latency by 5.6× and cut manual operations by 73% (§4.4).

Logical unification. All sources, stages, and derived features appear as columns of one logical table. Users operate via *Ingestion* (add rows with batch registration) and *Feature Engineering* (add columns declaratively with automated dependencies and lineage).

Physical separation. The Catalog (§4.1) maps logical columns to physical *table families*, enabling autonomous split/merge/materialization to handle small files, column limits, and mixed workloads without changing logical semantics.

3.2 Logical Data Model

OmniTable's logical data model is built around four principles: global primary key alignment, stage-aware data evolution, unbounded feature column expansion, and batch-level governance. Each row represents a traceable data entity; each column represents either the entity's state at a processing stage or a derived feature value. The unified schema is illustrated by the following DDL:

Listing 1: OmniTable logical schema (illustrative).

```
CREATE TABLE OmniTable (
  -- Global Primary Key & Lineage Anchor
  _ai_unique_id_ STRING NOT NULL,
  -- Ingestion Batch / Source Tag
  _ai_append_name_ STRING NOT NULL,

  -- Core Data Columns (Stage-aware)
  RawData        STRUCT<...>,
  ProcessedData  STRUCT<...>,
  TrainableData  STRUCT<...>,

  -- Dynamically Growing Feature Columns
  Feature_1      <Type1>,
  ...
  Feature_N      <TypeN>
);
```

`_ai_unique_id_` serves as the global primary key for alignment across sources, stages, and features, enabling feature backfill joins, point lookups, and full lineage auditing. The three core column groups—`RawData`, `ProcessedData`, and `TrainableData`—represent the original payload, cleaned intermediate representation, and trainable form respectively, with optional version suffixes (e.g., `ProcessedData_v2`) to capture processing iterations as versioned columns.

Feature columns `Feature_1`...`Feature_N` accommodate derived information such as quality scores, domain labels, compliance flags, and deduplication signatures. Explicit columnar representation enables features to be registered, versioned, dependency-resolved, and automatically refreshed, directly serving filtering, statistics, and export.

`_ai_append_name_` identifies the batch, source, and version of ingested data, making the batch the basic unit of governance for auditing, task splitting, and query pruning. The DDL above describes a logical contract, not a physical constraint: the Catalog maintains a logical-to-physical mapping, enabling physical splitting, materialization, and indexing transparently (§4.5).

3.3 System Architecture

As shown in Figure 2, OmniTable employs a service-oriented architecture centered on the Catalog, consisting of five core services that collaborate to form an end-to-end closed loop of ingestion (add rows), feature backfill (add columns), exploration (query and export), and governance (background optimization).

Catalog Service. The Catalog serves as the authoritative metadata plane and consistency decision point. It maintains the logical wide-table schema, logical-to-physical mappings, feature definitions (including column-level dependency DAGs), and registries for indexes and materialized views. All frontend operations and backend optimizations coordinate through the Catalog, decoupling view stability from physical layout evolvability.

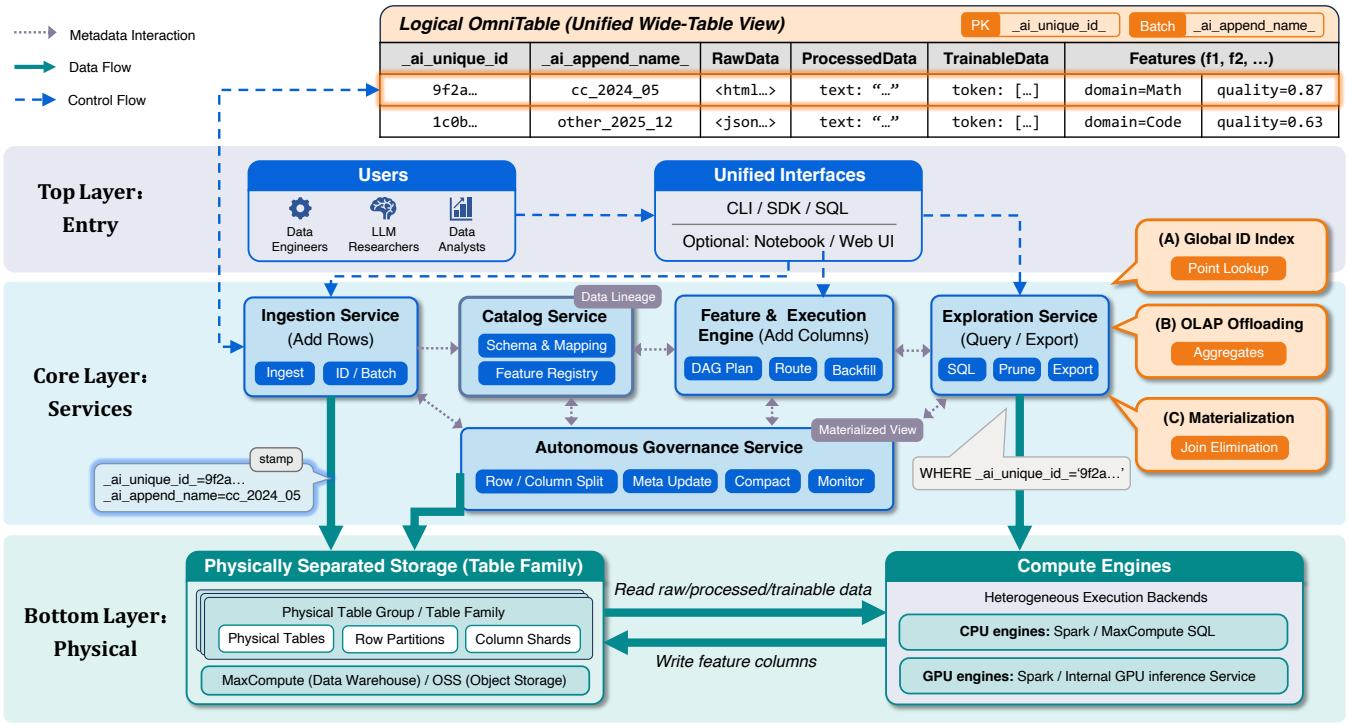

Figure 2: OmniTable system architecture. The Catalog Service acts as the central metadata plane, coordinating Ingestion (add rows), Feature & Execution Engine (add columns), Exploration Service (query and export), and Autonomous Governance Service (background optimization). The bottom layer shows physically separated storage and heterogeneous compute engines.

Ingestion Service. This service provides a unified row-addition entry point for multi-source heterogeneous data (MaxCompute tables, OSS [41] files). Beyond data transfer, it performs field mapping, generates the global primary key `_ai_unique_id_`, tags each record with `_ai_append_name_`, and atomically registers batch metadata in the Catalog, transforming discrete datasets into governed asset units.

Feature & Execution Engine. The engine transforms declarative feature definitions into physical execution plans. It parses dependencies via the Catalog, constructs a minimum-closure execution DAG in topological order, and intelligently routes tasks to heterogeneous backends—Spark/MaxCompute SQL [11, 73] for CPU-side ETL and UDF computation, or the GPU inference platform for model scoring. During execution, the engine manages concurrency control, checkpoint resumption, failure retries, and result write-back, then synchronizes column status, version, and lineage updates to the Catalog.

Exploration Service. This service handles mixed workloads including interactive exploration, anomaly backtracking, and training data export. It translates SQL on logical wide tables into physical execution plans using Catalog metadata, applying predicate pushdown [47], column pruning [1], and batch pruning. Three complementary acceleration paths are selected automatically: a global ID index for second-level point lookups on `_ai_unique_id_`, transparent OLAP offloading (via ClickHouse [55]) for aggregation queries, and background materialized views to eliminate runtime JOINs in filtered exports.

Autonomous Governance Service. This background service maintains long-term performance without user intervention. It monitors storage patterns (small-file density, partition skew, column growth) and query patterns (hot column combinations, frequent filters), then triggers transactional background optimizations—small-file compaction, row/column splitting, and materialized view construction [48]. All changes are isolated from frontend operations through Catalog versioning and registered back upon completion, allowing subsequent queries to transparently benefit.

4 SYSTEM DESIGN AND IMPLEMENTATION

This section details the internal design and key implementation of each core service, organized by the data lifecycle: Catalog (§4.1), Ingestion (§4.2), Feature & Execution Engine (§4.3), Exploration (§4.4), and Autonomous Governance (§4.5).

4.1 Catalog Service

The Catalog Service is the authoritative metadata plane for OmniTable. All frontend operations (ingestion, feature backfill, query export) and backend optimizations (splitting, merging, materialization) use the Catalog as a unified coordination point: frontend services obtain logical-to-physical mappings to generate correct

execution plans, while backend services register optimization results so that frontend queries transparently benefit. This design decouples logical view stability from physical layout evolvability.

4.1.1 Logical-Physical Schema Mapping. Users see a single logical wide table with a unified schema, but this table is physically backed by multiple tables—a *Table Family*. The Catalog maintains a five-layer entity model spanning LogicalTable, LogicalColumn, PhysicalTableGroup, PhysicalTable, and PhysicalColumn. A LogicalTable corresponds to a complete user-facing wide table (e.g., `aidata://tables/web`). Each LogicalColumn carries semantic information (name, type, analysis type, comments). A PhysicalTableGroup is an intermediate layer grouping physical tables by function or batch. Each PhysicalTable resides on the underlying engine (MaxCompute or OSS) with attributes such as storage path, partition information, and file format. PhysicalColumns are linked to LogicalColumns through bidirectional references in the Catalog.

The key advantage of this multi-layer mapping is physical layout evolvability. When the backend governance service (§4.5) detects that a physical table exceeds the storage engine's column limit (e.g., MaxCompute's ~1200-column limit), the system automatically performs column splitting by updating only the affected mapping pointers in the Catalog, without modifying any logical schema. Similarly, small-partition merges atomically update PhysicalTable entries after data rewrite. The query optimizer always obtains the latest mapping to generate plans, making physical changes fully transparent. Each mapping entry carries a version number and timestamp for historical backtracking. The Catalog stores metadata in a relational database with optimistic locking for concurrent update consistency; metadata query latency remains at the millisecond level in production.

The Catalog also manages batch (Append) metadata. Each ingestion registers an Append entry recording the batch name (`_ai_append_name_`), associated PhysicalTable identifiers, row count, ingestion time, and source information. This makes `_ai_append_name_` a key dimension for batch pruning, incremental backfill scheduling, and audit backtracking: queries filtering on a specific batch (e.g., `WHERE _ai_append_name_ = 'cc'`) can directly locate the corresponding physical table subset, significantly reducing I/O.

Listing 2: Feature definition schema (simplified).

```
FeatureDefinition {
  inputColumns: [col_1, col_2, ...],
  outputColumns: [{
    name: <column_name>, type: <data_type>,
    analysisType: <semantic_type>,
    comment: <description>
  }, ...],
  defaultFeatExpression: <expr>,
  computeEngine: <engine_preference>,
  tablePath: <wide_table_path>,
  extraInfo: { hints: <config>,
    parallel: <N>, where_condition: <filter> }
}
```

4.1.2 Feature Metadata and Lineage. OmniTable treats features as first-class metadata [34]. Complete feature definitions---computation logic, input/output dependencies, runtime preferences, and version history—are atomically registered in the Catalog. This centralizes feature computation logic that is traditionally scattered across codebases and enables automated lineage tracing and dependency resolution.

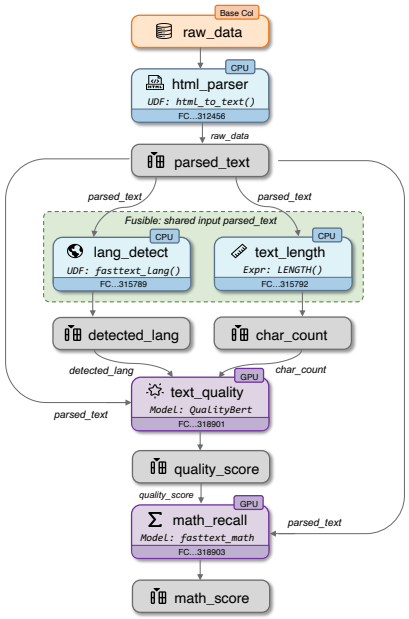

Figure 3: Feature computation DAG automatically constructed by the Catalog, showing column-level dependencies among registered features on the web wide table.

A feature is registered via the `omni-cli add feature` command with a structured definition specifying input columns, output columns (with name, type, and semantic annotation), computation expression, engine preference, and optional runtime hints (Listing 2).

Based on registered definitions, the Catalog automatically constructs a column-level dependency DAG. Figure 3 illustrates a real subgraph from the web wide table: `raw_data` → `html_parser` → `parsed_text`, which fans out to `lang_detect` and `text_length`; `quality_score` depends on both `parsed_text` and `detected_lang`; and `math_recall` depends on `parsed_text` and `quality_score`. The DAG is stored as an adjacency list, with each node recording its feature ID, column ID, and parent column IDs.

4.2 Ingestion Service

The Ingestion Service integrates heterogeneous corpora into the logical wide table through a declarative interface (`omni-cli submit`). Users specify source type, location, target wide table, and column mapping (e.g., `body_text:raw_data`). Unmapped source columns are ignored; missing logical columns are filled with `NULL` for subsequent backfill. Internally, a three-stage pipeline handles source adaptation (format parsing and validation), schema normalization, and physical writing (columnar organization for bulk loads, fast append for incremental writes). New wide tables are created on demand.

`_ai_unique_id_` serves as the global primary key. The default strategy uses deterministic content hashing (MD5 of `raw_data`), providing natural deduplication without coordination. A random

Algorithm 1: Declarative Feature Backfill

Input: BackfillPlan $P = \{tablePath, \{(a_i, c_j)\}\}$; Feature DAG G from Catalog
Output: Computed feature columns; updated Catalog metadata

```
/* Stage 1: Dependency Resolution                                   */
Q ← copy of P.pairs;
foreach (a_i, c_j) ∈ Q do
    Ancestors ← G.transitiveClosure(c_j);
    foreach f_k ∈ Ancestors do
        if Catalog.status(a_i, f_k) ≠ COMPUTED then
            Q ← Q ∪ {(a_i, f_k)};
D ← topologicalSort(Q, G);
/* Stage 2: Physical Plan Generation                                */
foreach (a_i, c_j) ∈ D in topological order do
    (P_in, P_out) ← Catalog.resolvePhysical(a_i, c_j);
    engine ← RouteEngine(c_j.computeEngine, c_j.mode);
    params ← AdaptiveTune(c_j, a_i, engine);
    T_ij ← buildTask(P_in, P_out, c_j.expr, engine, params);
    if ∃ (a_i, c_m) ∈ D sharing P_in and engine then
        T_ij ← fuseOperators(T_ij, T_im);
/* Stage 3: Scheduled Execution                                     */
while ∃ unfinished node in D do
    T_ij ← next ready task;
    Dispatch T_ij with UDF-level fault tolerance;
    if T_ij succeeds then
        Write results; record checkpoint;
    else
        if retries < max then Re-enqueue;
        else Mark FAILED; log to ErrorTable;
/* Stage 4: Metadata Commit                                         */
foreach completed T_ij do
    Catalog.atomicUpdate(a_i, c_j): status, physicalMapping,
      lineage;
```

UUID strategy is available when content deduplication is unnecessary (e.g., SFT data). Ingestion uses write-then-commit semantics: parallel shards process data, then the service atomically registers an Append entry in the Catalog recording batch name, physical tables, row count, source, and mapping. Data becomes visible only after commit; failures trigger rollback. `_ai_append_name_` serves as scheduling granularity for backfill (§4.3), query pruning (§4.4), and governance (§4.5). In production, the web wide table has ingested over 25 PB across 200+ batches.

4.3 Feature and Execution Engine

The Feature & Execution Engine shifts feature engineering from manual, table-by-table pipeline orchestration to declarative operations on logical wide tables. Users declare the computation logic of a feature; the system automatically resolves dependencies, generates execution plans, routes tasks to heterogeneous backends, and manages fault tolerance.

4.3.1 Declarative Feature Lifecycle. The user interaction reduces to two operations: registering a feature definition via `omni-cli add feature` (§4.1) and submitting a backfill plan via `omni-cli add jobs`. The system then executes the four-stage process shown in Algorithm 1.

Stage 1: Dependency Resolution. The user's backfill plan typically specifies only the target feature and batch (e.g., compute `math_recall` on batch `cc`). The system traverses the feature DAG to collect the transitive closure, checks each predecessor's computation status, and auto-expands the plan to include all uncomputed dependencies. Formally, given a user-specified set of backfill targets $P \subseteq \mathcal{A} \times C$ (append–column pairs) and the feature dependency DAG $G = (C, E)$, the minimum-closure backfill set is:

$$Q^* = \Big\{(a_i, c_k) \;\Big|\; (a_i, c_j) \in P,\;\; c_k \in \mathrm{Anc}_G(c_j) \cup \{c_j\},\;\; \mathrm{status}(a_i, c_k) \neq \textsc{Computed}\Big\} \tag{1}$$

where $\mathrm{Anc}_G(c_j)$ denotes the set of all ancestor nodes of c_j in G. Topological sorting of Q^* then produces an execution DAG respecting dependency order.

Stage 2: Physical Plan Generation. For each node in the DAG, the system resolves input/output physical tables via the Catalog, selects the execution engine based on the feature's `computeEngine` and operator characteristics, and sets resource parameters via adaptive tuning (§4.3.3). An important optimization is *operator fusion*: when multiple features in the same batch share input columns and target engine (e.g., `lang_detect` and `text_length` both reading `parsed_text` on Spark), the system merges them into a single SQL task computing multiple output columns in one scan, reducing redundant I/O by several times in production.

Stage 3: Scheduled Execution. The scheduler dispatches ready tasks (all parents completed) according to topological order and user-specified concurrency. Each task runs with UDF-level fault tolerance (§4.3.3). On success, results are written and partition-level checkpoints are recorded, supporting resumption after interruption. Failed tasks are retried up to a configured limit; beyond that, they are marked failed with details logged to an error table.

Stage 4: Metadata Commit. Upon completion, the system atomically updates the Catalog for each `(append, column)` pair: setting status to COMPUTED, registering the output physical mapping, and recording lineage (feature ID, version, timestamp, engine). This transforms computation results from temporary output into queryable, traceable assets.

4.3.2 Intelligent Routing for Heterogeneous Compute. Feature computation spans rule-based statistics (CPU-suited), deep learning scoring (GPU-required), and Python UDFs with complex dependencies (e.g., fastText [36], BERT [27]). Users declare `--mode=cpu_only|gpu_only|all` during registration; the system routes at runtime based on operator profile (UDF dependency analysis), engine capabilities (Spark/MaxCompute SQL [11] vs. GPU inference), and cluster load [15]. In production, ~70% of features route to CPU clusters, ~20% (fastText/BERT) are placed dynamically based on model size and resource conditions, and ~10% (e.g., Qwen [70] PPL) route to GPU. Users remain unaware of engine selection.

4.3.3 Elasticity and Fault Tolerance. PB-scale feature backfill faces two systemic risks: improper resource configuration causing OOM or resource waste, and bad data (e.g., extremely long texts, encoding

errors) causing entire batch tasks to fail. OmniTable addresses both through *adaptive parameter tuning* and *UDF-level fault tolerance*.

Adaptive Parameter Tuning. The engine computes resource parameters from two signals. First, heuristic rules use Catalog statistics (partition count, row count, data volume, average record size) and feature profiles: Python model loading raises Spark's `memoryOverhead`, while high partition counts adjust `mapper.split.size`. Second, History-Based Optimization (HBO) [58] records CPU time, peak memory, I/O throughput, feature identifiers, and batch sizes, then interpolates from similar historical runs. This strategy gives reasonable cold-start parameters and improves with history; in production it raised novice users' first-submission success rate from about 60% to over 90%.

UDF-Level Fault Tolerance. PB-scale unstructured data inevitably contains bad records, and one such record can fail a Spark task containing millions of inputs. OmniTable wraps each UDF invocation with timeout and memory checks. On Python OOM, timeout, or uncaught exception, it logs the record's `_ai_unique_id_`, exception type, and stack summary to an error table, writes NULL, and continues. The Catalog records success/failure counts and error-log references after completion, while configurable retries handle transient failures. Adaptive tuning reduces configuration failures before submission; UDF-level isolation contains data failures during execution.

4.4 Exploration Service

The Exploration Service bridges the gap between the logically unified wide-table view and physically distributed storage, handling mixed workloads: second-level point lookups for anomaly tracing, aggregation queries for feature distribution analysis, and high-throughput filtered exports (≥20 TB/h) for ablation experiments. It translates user SQL on logical wide tables into physical execution plans and selects among three complementary acceleration paths.

4.4.1 Logical Query Translation and Rewriting. The query optimizer translates user SQL on logical wide tables into plans executable on underlying physical table families. Because logical columns may span multiple physical tables (due to column splitting or batch-based organization), translation involves multi-step reasoning beyond simple table name replacement.

The optimizer parses user SQL to extract referenced columns and predicates, then queries the Catalog for each column's physical location. The result is typically a multi-table JOIN on `_ai_unique_id_` with UNION ALL across batches. The optimizer then applies standard rewrite rules: *predicate pushdown* [47] pushes filters into physical table scans; *column pruning* [1, 61] reads only required columns, avoiding I/O on large unused columns such as `raw_data`; and *batch pruning* uses Catalog Append metadata to skip physical tables of unrelated batches when `_ai_append_name_` filters are present.

After these optimizations, the optimizer attempts to match one of three acceleration paths: global ID index for equality lookups on `_ai_unique_id_` (§4.4.2), OLAP offloading for aggregate queries (§4.4.3), or materialized view matching for multi-column filtered exports (§4.4.4).

4.4.2 Global ID Index. Sample tracing retrieves a complete record by `_ai_unique_id_`, including all stage data and feature values. Without an index, it requires full scans across physical tables and takes minutes at PB scale.

OmniTable builds a global secondary index on `_ai_unique_id_` using a distributed key-value store (HBase [4, 19]). Each entry maps an ID to its physical table identifier, partition path, and row group offset. Index construction is asynchronous: incremental updates are triggered after successful ingestion or feature backfill commits, with version and coverage information maintained in the Catalog.

When the optimizer detects an equality filter on `_ai_unique_id_`, it routes to the index path. The index service returns physical locations with millisecond latency; the system reads only target row groups and assembles columns from relevant tables in parallel. In production, the index covers over 300 billion records, with single-ID latency of 5–15 seconds.

4.4.3 Analytical Query Offloading to OLAP. Analytical queries with aggregations are common during filtering strategy design but slow on batch engines (minutes to hours). OmniTable maintains a ClickHouse [55] instance, incrementally synchronizing frequently accessed feature columns based on query pattern analysis. Offloading is transparent: when the optimizer verifies all referenced columns are synchronized with sufficient freshness, it rewrites and routes to ClickHouse; otherwise it falls back to the batch engine. In production, this reduces aggregation response times from minutes to seconds (∼100×).

4.4.4 Join Elimination via Background Materialization. When query columns span three or more physical tables, runtime JOINs become the dominant bottleneck—each JOIN involves large-scale scans and expensive shuffles. For filtered export queries (typically involving 10+ feature columns), multi-table JOIN overhead reduces throughput well below the 20 TB/h target.

OmniTable eliminates runtime JOINs through background materialization [30, 32, 33]. The Autonomous Governance Service records column co-occurrence in query logs and periodically constructs a co-occurrence frequency matrix [52]. When a column combination exceeds a frequency threshold and spans multiple physical tables, it becomes a materialization candidate. The service pre-joins these columns on `_ai_unique_id_` into a materialized wide table and atomically registers its metadata (covered columns, batch range, version) in the Catalog.

At query time, the optimizer checks whether a materialized view fully covers the required columns. If so, the query is rewritten as a single-table scan, eliminating all JOINs. For partial coverage, the optimizer evaluates whether using the view reduces JOINs (e.g., from three to one) and selects a mixed path accordingly.

Materialized views are maintained via incremental refresh triggered by Catalog change events; delayed refreshes are handled by compensation queries on uncovered increments merged via UNION ALL. In production, ∼80% of filtered export queries reference 15–20 feature columns across 3–5 physical tables; after materialization, these reduce to single-table scans with 3–5× latency improvement and throughput above 20 TB/h.

4.5 Autonomous Governance Service

Without continuous maintenance, PB-scale continuous writes and growing column counts inevitably degrade performance: small files

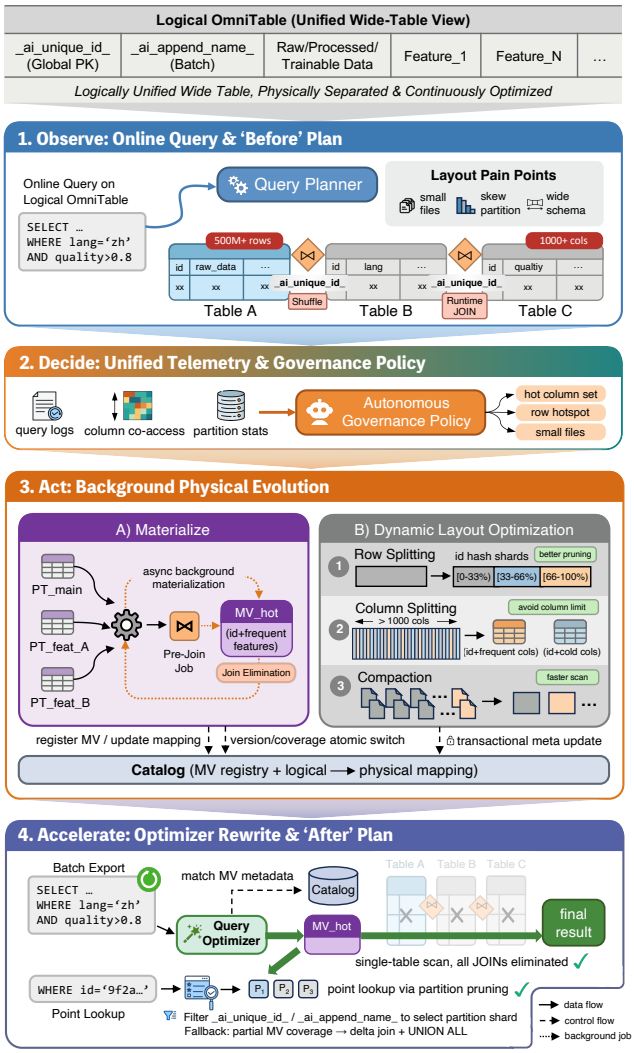

Figure 4: Autonomous governance pipeline: observe layout pain points, decide optimization policy, perform background physical evolution (materialization, splitting, compaction), and accelerate queries via optimizer rewriting.

reduce scan efficiency [29], column counts hit engine limits [61], and partition skew causes long-tail queries. The Autonomous Governance Service addresses these challenges through automated background optimization, isolated from frontend operations.

4.5.1 Transactional Consistency Guarantees. All backend optimizations follow a three-phase Prepare-Execute-Commit protocol [13]. In the *Prepare* phase, the service acquires an exclusive lock at PhysicalTableGroup granularity from the Catalog; if the target is occupied by a frontend write, the optimization is postponed. Frontend reads do not hold exclusive locks but obtain a consistent metadata snapshot at query start, ensuring in-flight queries are unaffected by concurrent backend changes [3].

In the *Execute* phase, physical reorganization (merging, splitting, materialization) writes new files to a staging area without modifying existing files. In the *Commit* phase, the service atomically updates the Catalog: registering new physical files, updating logical-to-physical mappings, and marking old files for reclamation. The new layout becomes visible only after successful commit; subsequent queries automatically use the optimized layout. Failures trigger metadata rollback and staging cleanup.

4.5.2 Dynamic Storage Layout Optimization. The physical layout continuously adapts to data growth, feature evolution, and query pattern changes through three mechanisms (Figure 4).

Row Splitting. When a physical partition exceeds thresholds (e.g., 500M rows or 500 GB), the system divides it into N sub-partitions based on `_ai_unique_id_` hash ranges. The Catalog records each sub-partition's range, enabling precise partition pruning for ID-based queries and improved parallelism for full scans.

Column Splitting. When logical column count approaches the storage engine's limit (e.g., MaxCompute's ~1200 columns), the governance service groups columns by access frequency from query logs. Low-frequency feature columns are migrated to auxiliary physical tables (Column Shards) containing only `_ai_unique_id_` and the migrated columns. The main table retains high-frequency and core columns. The Catalog updates logical-to-physical mappings; queries involving only high-frequency columns require no JOINs. In production, the web wide table's 800+ logical columns are distributed across 4–6 physical tables, each with 200–300 columns.

Small-File Compaction. Append-only writes from ingestion and backfill accumulate small files. A background compaction service periodically scans file size distributions and triggers merging when thresholds are exceeded (e.g., >1000 files per partition or >50% files smaller than 64 MB). The compaction task reads small files, reorders data by `_ai_unique_id_`, and writes optimally sized files (256 MB–1 GB) with columnar encoding and compression [28]. OmniTable uses a *tiered compaction strategy* [46]: daily lightweight merges for hot partitions and weekly deep merges for cold partitions, balancing merge benefits against resource costs (approximately 5–8% of cluster resources in production).

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Datasets and Production Environment. All experiments were conducted on a real production deployment of OmniTable. Table 1 summarizes the core data assets. The web wide table (`web`) is the largest, covering 13 sources (Common Crawl, internal crawls, licensed third-party corpora, etc.) with over 25 PB, 300B+ records, 800+ logical columns (200+ registered features), 6 physical tables, and 200+ ingested batches. The code (`code`), PDF (`pdf`), and SFT (`post_sft`) wide tables cover the remaining major data types, managing a combined total of over 35 PB.

Workload Characteristics. We evaluate three groups: (1) governance batch tasks (ingestion, feature backfill, failure recovery); (2) exploratory queries (point lookups, filtered exports, aggregation statistics); and (3) scalability stress tests. The 200+ features on `web` span ~70% lightweight SQL/rule UDFs suited for CPU/Spark [71, 73], ~20% CPU-based model inference UDFs (fastText [36], BERT [27]), and ~10% GPU-intensive operators (Qwen [70] PPL).

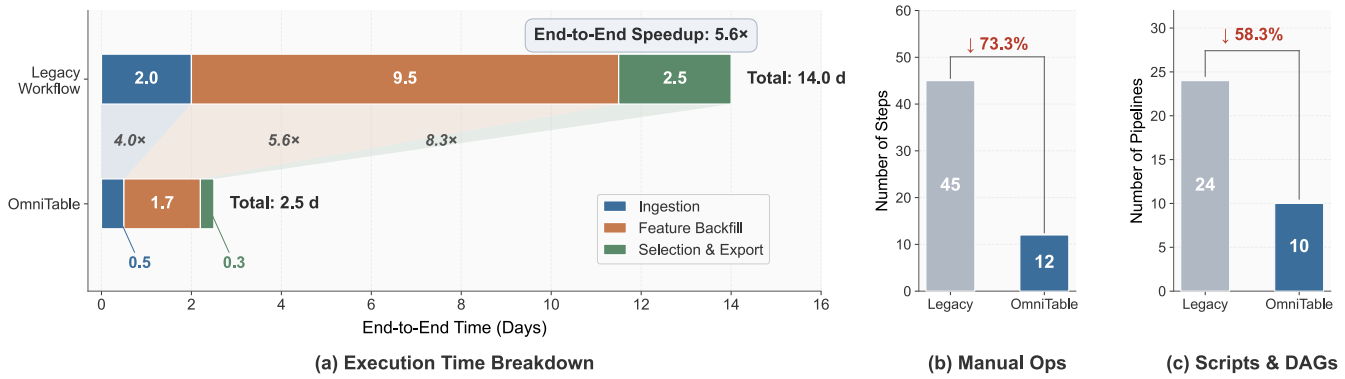

Figure 5: End-to-end SFT data curation comparison. (a) Execution time breakdown by stage: OmniTable achieves 5.6× speedup (14.0 d → 2.5 d). (b) Manual operation steps reduced by 73%. (c) Independent pipelines/scripts reduced by 58%.

Table 1: Core data assets used in the experiments.

Table	Records	Size	Log. Cols	Phys. Tbl.	Batches
web	300B+	25 PB	800+	6	200+
code	3.4B	3.8 PB	350	4	85
pdf	1.8B	5.2 PB	280	3	62
post_sft	210M	0.8 PB	120	3	45
Total	**305B+**	**35 PB**	—	**16**	**392+**

Hardware. CPU side: MaxCompute/Spark hybrid cluster with ~12,000 nodes (64-core CPU, 256 GB RAM each). GPU side: ~300 NVIDIA L20 cards (48 GB VRAM). Storage: MaxCompute warehouse and OSS object storage using Parquet [5] columnar format. Catalog: high-availability MySQL cluster. ID index: HBase [4] cluster. OLAP: ClickHouse [55] cluster. Spark task parameters are set automatically by adaptive tuning unless noted otherwise.

Baseline (Legacy Workflow). We defined a baseline replicating the pre-OmniTable production workflow (§2): (1) data organized in independent MaxCompute tables without unified schema; (2) feature engineering via manual pipeline orchestration on a visual canvas; (3) cross-dataset queries via manually written multi-table JOINs; (4) task tuning via engineer experience. Efficiency data is derived from historical task records and operation logs; one manual retry is allowed per failure for fairness. This comparison targets *workflow-level coordination cost*—the overhead of locating tables, wiring pipelines, diagnosing failures, and re-submitting jobs—which is orthogonal to compute-level optimizations in systems such as Data-Juicer [21] or Delta Lake [9]. Per-subsystem performance is validated separately in §5.3–5.5.

Ablation (OmniTable-NoGov). To isolate the contribution of autonomous governance, we disable background compaction, row splitting, and column splitting, retaining only logical query translation and basic pruning. This quantifies the necessity of physical layout evolution at PB scale (§5.3).

Methodology. All time results are medians of three runs. Exploration queries are replayed at fixed concurrency (20) within the same time window; error bars show cross-run variation.

5.2 End-to-End Data Curation Efficiency

We compare OmniTable and Legacy Workflow on a real SFT data processing scenario: collecting instruction data from 8 sources, computing 12 features (9 CPU UDFs, 3 GPU inference) covering quality scoring, safety compliance, and domain classification, and exporting a high-quality subset for ablation experiments. This scenario covers all three stages—ingestion, feature backfill, and filtered export. Input batches and filtering thresholds are fixed to ensure identical business objectives.

Legacy Workflow. Ingestion takes ~2 days to locate 8 source tables, write conversion scripts, and copy data. Feature backfill is the main bottleneck at ~9.5 days: engineers configure 12 feature nodes × 8 tables (~96 nodes) on the orchestration canvas, repeatedly handling failures from improper parameters or bad data. Without UDF-level fault tolerance, a single anomalous record fails the entire TB-scale batch, triggering costly investigate-retry cycles. Filtered export takes ~2.5 days writing manual multi-way JOINs across 8 result tables. Total: ~14 days, ~45 manual steps, 24 pipelines, 35 physical tables.

OmniTable. Ingestion takes ~0.5 days via 8 `omni-cli submit` commands. Feature backfill takes ~1.7 days: after registering 12 feature definitions, a single backfill plan triggers automatic dependency resolution, heterogeneous engine routing, adaptive tuning, and UDF-level fault tolerance—no manual pipeline orchestration required. Operator fusion merges features sharing input columns into single SQL tasks. Filtered export takes ~0.3 days via a single SQL query on the logical wide table with automatic materialized view matching. Total: ~2.5 days, ~12 manual steps, 10 commands, 1 logical wide table.

Results. Figure 5 shows that OmniTable achieves a 5.6× end-to-end speedup over the Legacy Workflow. The feature backfill stage contributes the largest gain (9.5 d → 1.7 d, 5.6×), driven by declarative lifecycle management eliminating per-table orchestration and adaptive fault tolerance eliminating manual failure recovery. Manual steps decrease by 73.3% (45 → 12), pipelines by 58.3% (24 → 10), and engineers are freed from overnight pipeline monitoring.

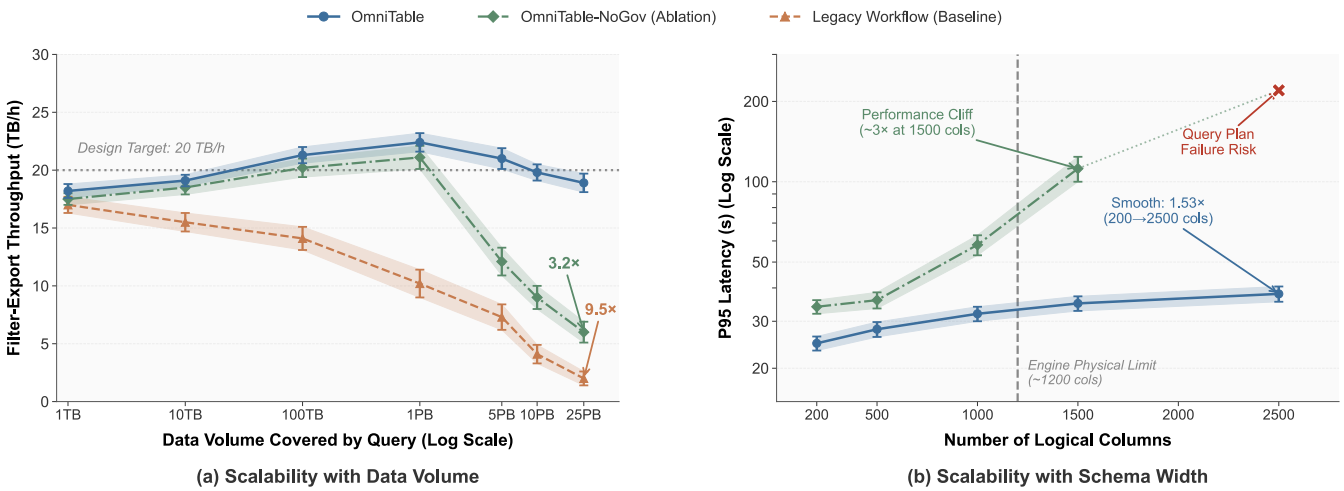

Figure 6: Scalability evaluation. (a) Filter-export throughput vs. data volume (1 TB–25 PB). OmniTable maintains 18–23 TB/h; OmniTable-NoGov degrades beyond 1 PB; Legacy Workflow drops to ~2 TB/h at 25 PB (9.5× slower). (b) P95 query latency vs. number of logical columns (200–2500). OmniTable's column splitting transparently crosses the ~1200-column engine limit (1.53× over full range); NoGov hits a ~3× performance cliff at 1500 columns and query failures beyond.

5.3 PB-Scale Performance and Scalability

This section evaluates query performance as data volume and schema width grow, validating the effectiveness of autonomous governance (compaction, row splitting, column splitting). We compare OmniTable, OmniTable-NoGov, and Legacy Workflow.

Experimental Design. Two controlled experiments on web: (1) *Data volume scalability*: a fixed 15-column filtered export query with 8 predicates on 12 feature columns; data coverage varied from 1 TB to 25 PB by selecting different batch combinations. (2) *Schema width scalability*: fixed ~2 PB coverage; logical columns increased from 200 to 2500 by adding lightweight features; a fixed 10-column interactive filter query returning 1000 rows, ensuring the query column set is constant regardless of total column count.

Data Volume Scalability (Figure 6a). OmniTable maintained stable throughput of 18–23 TB/h from 1 TB to 25 PB, due to the synergy of three mechanisms: small-file compaction keeps files in the 256 MB–1 GB optimal range for sequential I/O; batch pruning via Catalog metadata skips irrelevant physical tables; and row splitting bounds per-partition size for linear parallelism scaling.

OmniTable-NoGov shows comparable throughput below 1 PB but declines sharply beyond: ~10 TB/h at 5 PB (3.2× gap vs. OmniTable) and ~5 TB/h at 25 PB, due to accumulated small files degrading I/O and unsplit partitions creating long-tail tasks. Legacy Workflow degrades most severely (~2 TB/h at 25 PB, 9.5× slower than OmniTable), because cross-table queries require runtime multi-way JOINs with shuffle costs scaling linearly with data volume.

Schema Width Scalability (Figure 6b). With column splitting enabled, OmniTable's P95 latency increased from ~25 s to ~38 s (1.53×) as columns grew from 200 to 2500—notably crossing MaxCompute's ~1200-column physical limit with no performance discontinuity. Column splitting automatically migrates low-frequency columns to auxiliary tables based on access frequency; since the fixed query involves only high-frequency columns residing in the main table, it completes via single-table scan without JOINs.

OmniTable-NoGov shows similar latency below 800 columns but exhibits a sharp degradation near the engine limit: P95 rises to ~55 s at 1000 columns and ~110 s at 1500 columns (~3× OmniTable). Beyond 1200 columns, some queries fail entirely due to exceeding the physical column limit, requiring manual table redesign.

Summary. OmniTable demonstrates stable performance across four orders of magnitude in data volume (1 TB → 25 PB) and exceeding the engine's physical column limit by ~2× (200 → 2500 columns). The ablation study confirms that autonomous governance is indispensable: without it, throughput degrades by >3× at PB scale, and schema growth causes performance cliffs and query failures near engine limits.

5.4 Engine Intelligence and Elasticity

This section quantifies three key mechanisms of the execution engine through job-level controlled experiments on web. All results are medians of three replicates; resource consumption is measured in CPU Hours (cores × hours) including retry overhead.

Experiment 1: Operator Fusion. We selected 8 CPU/Spark features sharing `parsed_text` as input on a Common Crawl batch (~2.5 PB, 300B+ records), covering text length, language detection, n-gram repetition, URL-content ratio, digit proportion, garbage-character flags, and content MD5. *Fusion-Off* generates 8 independent tasks each scanning `parsed_text`; *Fusion-On* merges them into a single task producing 8 columns in one scan. As shown in Figure 7(a–b), Fusion-On reduces total CPU Hours by 55.9% (42K → 18.5K), with I/O read dropping from 27.3K to 7.4K CPU Hours (8× → 1× scans). End-to-end time decreases from 38 h to 14 h (2.7× speedup).

Experiment 2: UDF-Level Failover. We backfilled the fastText [36] math-recall feature `math_recall_v4` on a 500 GB batch

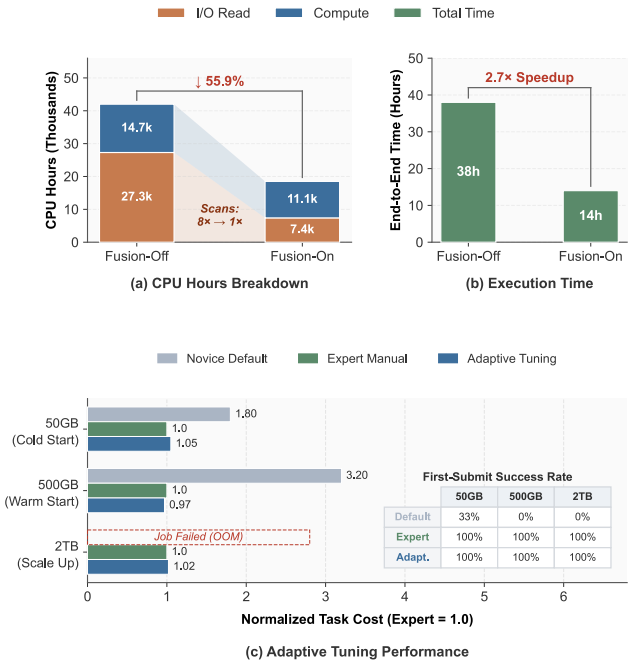

Figure 7: Execution evaluation. (a) CPU Hours breakdown: Fusion-On reduces total CPU Hours by 55.9% (scans: 8× → 1×). (b) End-to-end time: 2.7× speedup (38 h → 14 h). (c) Adaptive tuning normalized task cost (Expert = 1.0) across batch sizes; Default fails (OOM) at 2 TB. Inset: first-submit success rates.

Table 2: UDF-level failover comparison on `math_recall_v4` (500 GB, ~600M records, ~31K anomalous at 0.005%).

Config.	Success	Records OK	Anomalies	Time	Manual
Legacy	0%↛100%	600M	~31K (removed)	~52 h	3 rounds
Failover-Off	0% (failed)	0	—	Failed	≥1
Failover-On	100% (one-pass)	599.97M	31,247 (logged)	~6.2 h	0

(~600M records) containing ~31K anomalous records (0.005%) that trigger OOM due to extreme text length or encoding errors. We compared three configurations (Table 2): Failover-On completes 99.995% of records in one pass (~6.2 h, zero manual intervention), logging 31,247 anomalies to an error table. Failover-Off fails entirely. Legacy Workflow requires 3 manual investigate-remove-resubmit cycles (~52 h total, including ~18 h manual effort). This demonstrates that even at 0.005% anomaly rate, task-level failure granularity causes highly asymmetric costs; record-level isolation eliminates this.

Experiment 3: Adaptive Tuning. We backfilled `quality_score` (BERT-based [27], ~350 MB model, sensitive to `memoryOverhead`) on three batch sizes: 50 GB, 500 GB, and 2 TB. We compared Default (Spark defaults), Adaptive (OmniTable auto-tuning), and Expert (manual tuning by experienced engineers). The metric is Normalized Task Cost = CPU Hours × (1 + Retry Count), with Expert normalized to 1.0.

As shown in Figure 7(c), Adaptive achieves 100% first-submit success across all sizes with cost within 5% of Expert. Default fails at ≥500 GB (OOM). At 500 GB, Adaptive (0.97) slightly outperforms Expert (1.0) because HBO interpolation produces tighter parameters than human experts' conservative margins. The progressive strategy uses heuristic rules for cold starts (increasing `memoryOverhead` based on data profiling) and HBO for warm starts.

Combined Effect. In the SFT scenario of §5.2 (feature backfill 7.8 d saving), the three mechanisms contribute complementary savings: operator fusion saves ~1.5 d of I/O overhead; UDF fault tolerance avoids ~2–3 d of investigate-retry cycles; adaptive tuning eliminates ~1–2 d of parameter-related failures. These partially overlap, totaling ~5–6.5 d; the remainder comes from scheduling improvements.

5.5 Hybrid-Accelerated Data Exploration

This section evaluates the three complementary acceleration technologies in the Exploration Service on `web` (25 PB, 300B+ records, 800+ logical columns, 6 physical tables). All queries were replayed at concurrency 20; results are medians of three runs.

Point Query Performance (Figure 8a). With the global ID index enabled, single-record lookups by `_ai_unique_id_` achieve P50 latency of 8.3 s and P99 of 14.7 s. Without the index, the system must scan all physical tables that may contain the target ID, yielding P50 of 184 s and P99 of 612 s+—a speedup of 22.2× (P50) to 41.6×+ (P99). Even at P99, a complete logical row (800+ columns) is returned within 15 s, transforming anomaly backtracking from batch submission to interactive querying.

Analytical Query Acceleration (Figure 8b). OLAP offloading to ClickHouse [55] achieves 94–154× speedup across four representative queries (single-column COUNT, two-column GROUP BY, filtered COUNT, multi-column aggregation), reducing response times from minutes to under 10 s. ClickHouse's sparse index pruning yields the highest speedup on filtered queries. Coverage spans ~30 high-frequency columns; unsynchronized queries fall back to Spark.

Filter-Export Acceleration (Figure 8c). Background materialized views show increasing benefit as queries span more physical tables. S1 (5-column filter, 2 tables): MV-Off achieves 14.3 TB/h (single JOIN, acceptable shuffle cost); MV-On reaches 22.6 TB/h (1.6×). S2 (10 columns, 3 tables): MV-Off drops to 8.7 TB/h due to cascading JOINs; MV-On maintains 21.4 TB/h (2.5×). S3 (15 columns, 4 tables): MV-Off falls to 4.8 TB/h, well below the 20 TB/h target; MV-On rewrites the query as a single-table scan, sustaining 20.1 TB/h (4.2×). Materialized view benefit is positively correlated with the number of eliminated JOINs; typical ablation experiment queries (10–15 feature columns) fall in the S2–S3 range, making materialized views essential for meeting the throughput target.

Synergistic Effect. The three technologies cover complementary query patterns: ID index for point lookups (second-level latency), OLAP offloading for aggregation analysis (second-level response), and materialized views for multi-column filtered exports (≥20 TB/h). The query optimizer selects the optimal path automatically. Together, they reduce the data filtering phase in the SFT scenario (§5.2) from 2.5 days to ~0.3 days, covering the full query chain from anomaly identification through distribution analysis to data export.

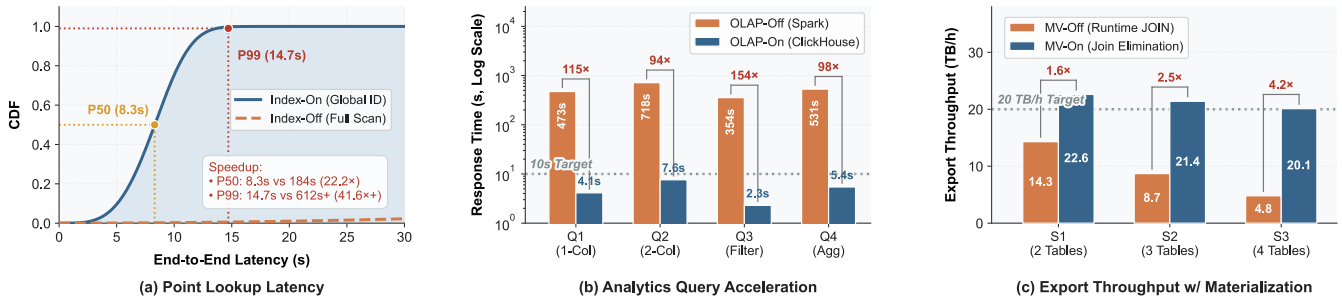

Figure 8: Hybrid acceleration for data exploration. (a) Point lookup latency CDF: global ID index achieves P50 = 8.3 s vs. 184 s full scan (22.2×), P99 = 14.7 s vs. 612 s+ (41.6×+). (b) Analytical query response time (log scale): OLAP offloading to ClickHouse achieves 94–154× speedup, all under 10 s. (c) Filter-export throughput: materialized views eliminate runtime JOINs, achieving 1.6–4.2× speedup and sustaining ≥20 TB/h across scenarios.

6 LESSONS LEARNED

Record-level fault tolerance is non-negotiable. Before UDF-level isolation, 30–40% of engineering hours were spent on failure investigation. A single malformed record among billions routinely failed entire multi-TB jobs. The 3–5% execution overhead of per-record wrapping eliminated our most expensive category of human intervention.

Autonomous governance is existential, not optional. We initially deferred background compaction and splitting as "nice-to-have." Within three months, small-file accumulation degraded query latency by 3–5× and column growth hit engine limits. Without continuous physical layout evolution, a PB-scale continuously-written system becomes unusable within weeks.

Adoption beats capability if engineers won't change habits. Algorithm engineers lived and breathed SQL: their instinct was to spin up yet another table and write yet another query. Asking them to abandon that habit upfront was a losing battle. Instead, we met them where they were: the logical wide table speaks plain SQL; a feature template library let them register a new feature in minutes rather than wiring a new pipeline; and lineage records are queryable tables, so auditing felt no different from `SELECT`ing a result. Only after these pain points disappeared did the deeper benefits—automated dependency resolution, heterogeneous routing, autonomous governance—become visible. A system that demands a workflow change before delivering value will be worked around, not adopted.

7 RELATED WORK

Lakehouse Architecture. Delta Lake [9], Apache Iceberg [7], and Apache Paimon [8] provide ACID, time travel, and schema evolution on data lakes; OmniTable can be viewed as a reusable layer above such storage. Lakehouse systems are largely *table-centric* and do not treat a *column/UDF dependency DAG* and *operator-aware CPU/GPU routing* as first-class lifecycle objects. OmniTable targets these LLM-specific needs via Catalog-managed feature metadata, dependency closure, routing, and autonomous layout optimization [10].

Feature Store. Feast [57], Tecton [62], and Hopsworks [24] manage ML features with emphasis on low-latency online serving of pre-computed structured features. OmniTable addresses a different problem: PB-scale backfill of UDF-defined features over raw unstructured text with CPU/GPU routing, governed across the full data lifecycle. These settings are sufficiently distinct that a shared benchmark does not exist; quantitative comparison would conflate compute efficiency with orchestration overhead.

Data Orchestration and Transformation. dbt [23] and Apache Airflow [6] operate at table and task granularity without domain-specific automation (dependency resolution, heterogeneous routing, UDF-level fault tolerance, adaptive tuning). OmniTable elevates operations to rows and columns on a logical wide table with these capabilities built in.

LLM Data Processing Frameworks. Data-Juicer [21], RedPajama [67], Dolma [60], and SmallPond [26] provide operator libraries for LLM data cleaning but follow the pipeline paradigm without unified cross-dataset management. OmniTable operates at a different layer: it provides orchestration, governance, and storage abstraction *above* operator libraries, complementing rather than competing with them—their operators can be directly registered as OmniTable UDFs.

ML Lifecycle and Data Management. TFX [12], Polyzotis et al. [51], Whang et al. [69], and Snorkel [53] motivate the data-centric perspective but do not provide the unified storage abstraction, declarative feature backfill, or autonomous governance that OmniTable contributes for PB-scale LLM scenarios.

8 CONCLUSION AND FUTURE WORK

OmniTable addresses heterogeneity, scale, agility, and traceability in PB-scale LLM data governance via *Logical Unification, Physical Separation*. It combines a unified wide-table abstraction, declarative feature lifecycle management, an adaptive execution engine, and hybrid acceleration to reduce exploration latency from minutes to seconds while sustaining ≥20 TB/h export throughput. In production, OmniTable manages over 35 PB of LLM training data, reducing the governance cycle from ~14 days to ~2.5 days (5.6×).

Future directions include intelligent data recommendation to close the Data-Centric AI loop [53, 69], batch-stream unification for near-real-time feature computation [18], and synthetic data governance with generation lineage tracking.

REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 967–980.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[3] R. Agrawal, M. J. Carey, and L. W. McVoy. 1987. The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems. *IEEE Transactions on Software Engineering* 12 (1987), 1348–1363.

[4] Apache Software Foundation. 2008. *Apache HBase Reference Guide*. Retrieved July 6, 2026 from https://hbase.apache.org

[5] Apache Software Foundation. 2013. *Apache Parquet*. Retrieved July 6, 2026 from https://parquet.apache.org

[6] Apache Software Foundation. 2015. *Apache Airflow*. Retrieved July 6, 2026 from https://airflow.apache.org

[7] Apache Software Foundation. 2018. *Apache Iceberg: An Open Table Format for Huge Analytic Datasets*. Retrieved July 6, 2026 from https://iceberg.apache.org

[8] Apache Software Foundation. 2023. *Apache Paimon: A Streaming Data Lake Platform*. Retrieved July 6, 2026 from https://paimon.apache.org

[9] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Van Hovell, A. Ionescu, A. Łuszczak, et al. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[10] M. Armbrust, A. Ghodsi, R. Xin, M. Zaharia, et al. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1383–1394.

[12] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[13] P. A. Bernstein and N. Goodman. 1981. Concurrency Control in Distributed Database Systems. *Comput. Surveys* 13, 2 (1981), 185–221.

[14] BigScience Workshop, T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, et al. 2022. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv preprint arXiv:2211.05100* (2022).

[15] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. 2015. The End of Slow Networks: It's Time for a Redesign. *arXiv preprint arXiv:1504.01048* (2015).

[16] A. Z. Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997*. IEEE, 21–29.

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[20] M. S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. 380–388.

[21] D. Chen, Y. Huang, Z. Ma, H. Chen, X. Pan, C. Ge, D. Gao, Y. Xie, Z. Liu, J. Gao, et al. 2024. Data-Juicer: A One-Stop Data Processing System for Large Language Models. In *Companion of the 2024 International Conference on Management of Data*. 120–134.

[22] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[23] C. Cyr and D. Dorsey. 2023. Introduction to dbt. In *Unlocking dbt: Design and Deploy Transformations in Your Cloud Data Warehouse*. Springer, 1–40.

[24] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, A. A. Ormenisan, S. Niazi, D. Bzhalava, K. Mak, V. Jouffrey, M. Ronström, R. Cunningham, et al. 2024. The Hopsworks Feature Store for Machine Learning. In *Companion of the 2024 International Conference on Management of Data*. 135–147.

[25] J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[26] DeepSeek. 2025. SmallPond: A Lightweight Data Processing Framework Built on DuckDB and 3FS. Retrieved July 6, 2026 from https://github.com/deepseek-ai/smallpond

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[28] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 29–43.

[30] J. Goldstein and P.-Å. Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *ACM SIGMOD Record* 30, 2 (2001), 331–342.

[31] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[32] A. Gupta, I. S. Mumick, et al. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin* 18, 2 (1995), 3–18.

[33] A. Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (2001), 270–294.

[34] P. Helland. 2015. Immutability Changes Everything. *Commun. ACM* 59, 1 (2015), 64–70.

[35] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. 2022. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556* 10 (2022).

[36] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. 2017. Bag of Tricks for Efficient Text Classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. 427–431.

[37] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372.

[38] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020).

[39] H. Laurençon, L. Saulnier, T. Wang, C. Akiki, A. Villanova del Moral, T. Le Scao, L. Von Werra, C. Mou, E. González Ponferrada, H. Nguyen, et al. 2023. The BigScience ROOTS Corpus: A 1.6TB Composite Multilingual Dataset. *arXiv preprint arXiv:2303.03915* (2023).

[40] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. 2022. Deduplicating Training Data Makes Language Models Better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8424–8445.

[41] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 331–346.

[42] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[43] Y. Liu, J. Cao, C. Liu, K. Ding, and L. Jin. 2025. Datasets for Large Language Models: A Comprehensive Survey. *Artificial Intelligence Review* 58, 12 (2025), 403.

[44] S. Longpre, R. Mahari, A. Chen, N. Obeng-Marnu, D. Sileo, W. Brannon, N. Muennighoff, N. Khazam, J. Kabbara, K. Perisetla, et al. 2024. A Large-Scale Audit of Dataset Licensing and Attribution in AI. *Nature Machine Intelligence* 6, 8 (2024), 975–987.

[45] N. Muennighoff, A. Rush, B. Barak, T. Le Scao, N. Tazi, A. Piktus, S. Pyysalo, T. Wolf, and C. A. Raffel. 2023. Scaling Data-Constrained Language Models. *Advances in Neural Information Processing Systems* 36 (2023), 50358–50376.

[46] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[47] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proceedings of the VLDB Endowment* 11, 11 (2018).

[48] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*, Vol. 4. 1.

[49] G. Penedo, H. Kydlíček, A. Lozhkov, M. Mitchell, C. A. Raffel, L. Von Werra, T. Wolf, et al. 2024. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. *Advances in Neural Information Processing Systems* 37 (2024), 30811–30849.

[50] G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, A. Cappelli, H. Alobeidli, B. Pannier, E. Almazrouei, and J. Launay. 2023. The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only. *arXiv preprint arXiv:2306.01116* (2023).

[51] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.

[52] J. R. Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (1986), 81–106.

[53] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 11. 269.

[54] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. 2017. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. *Amazon Science* (2017).

[55] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov. 2024. ClickHouse: Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[56] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems* 28 (2015).

[57] T. Sell and W. Pienaar. 2019. *Introducing Feast: An Open Source Feature Store for Machine Learning*. Retrieved July 6, 2026 from https://feast.dev

[58] P. Shankhdhar, F. Liu, J. Narale, J. Sun, R. Schlussel, and L. Antova. 2024. Presto's History-Based Query Optimizer. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4077–4089.

[59] D. Soboleva, F. Al-Khateeb, R. Myers, J. R. Steeves, J. Hestness, and N. Dey. 2023. SlimPajama: A 627B Token Cleaned and Deduplicated Version of RedPajama. *Hugging Face dataset* (2023). Retrieved July 6, 2026 from https://huggingface.co/datasets/cerebras/SlimPajama-627B

[60] L. Soldaini, R. Kinney, A. Bhagia, D. Schwenk, D. Atkinson, R. Authur, B. Bogin, K. Chandu, J. Dumas, Y. Elazar, et al. 2024. Dolma: An Open Corpus of Three Trillion Tokens for Language Model Pretraining Research. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15725–15788.

[61] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. 553–564.

[62] Tecton 2020. *Tecton: The Enterprise Feature Platform.* Retrieved July 6, 2026 from https://www.tecton.ai

[63] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. 2010. Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 996–1005.

[64] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[65] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. 2016. ModelDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–3.

[66] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 30 (2017).

[67] M. Weber, D. Fu, Q. Anthony, Y. Oren, S. Adams, A. Alexandrov, X. Lyu, H. Nguyen, X. Yao, V. Adams, et al. 2024. RedPajama: An Open Dataset for Training Large Language Models. *Advances in Neural Information Processing Systems* 37 (2024), 116462–116492.

[68] L. Weng, D. Liu, W. Zhu, R. Zhu, J. Zheng, B. Ding, Z. Zhang, and J. Zhou. 2026. Learned Query Optimizer in Alibaba MaxCompute: Challenges, Analysis, and Solutions. *arXiv preprint arXiv:2602.07336* (2026).

[69] S. E. Whang, Y. Roh, H. Song, and J.-G. Lee. 2023. Data Collection and Quality Challenges in Deep Learning: A Data-Centric AI Perspective. *The VLDB Journal* 32, 4 (2023), 791–813.

[70] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, B. Gao, C. Huang, C. Lv, et al. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[71] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[72] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi. 2024. The Shift from Models to Compound AI Systems. Retrieved July 6, 2026 from https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/

[73] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.

[74] G. Zhang, S. Qu, J. Liu, C. Zhang, C. Lin, C. L. Yu, D. Pan, E. Cheng, J. Liu, Q. Lin, et al. 2024. MAP-NEO: Highly Capable and Transparent Bilingual Large Language Model Series. *arXiv preprint arXiv:2405.19327* (2024).