A Comprehensive Analysis of Apache Doris Internals

Section 1: Executive Summary

Apache Doris is a modern, Massively Parallel Processing (MPP)-based real-time analytical data warehouse engineered to deliver sub-second query performance on massive datasets.[1] Since its donation to the Apache Software Foundation and subsequent graduation as a Top-Level Project, Doris has established itself as a formidable tool for a wide range of analytical applications, including high-concurrency reporting, ad-hoc analysis, and data lake query acceleration.[3] Its adoption by over 5,000 enterprises globally, including technology giants like TikTok and Tencent, underscores its capability to handle demanding production workloads.[1] This report provides a deeply technical exegesis of the internal mechanisms that underpin Apache Doris, dissecting its architecture, storage engine, query processing pipeline, and data management operations to reveal the design principles that enable its performance and scalability.

At the heart of Doris lies a deliberately simplified two-process architecture, comprising Frontend (FE) and Backend (BE) nodes. This highly integrated model is a cornerstone of its operational simplicity, significantly reducing the maintenance overhead often associated with complex, multi-component big data ecosystems.[1] The FE serves as the cluster's brain, managing metadata, coordinating nodes, and meticulously planning queries, while the BEs act as the workhorses, responsible for data storage and the physical execution of computational tasks.

A pivotal aspect of Doris's evolution is its architectural duality. While rooted in a traditional compute-storage coupled (Shared-Nothing) model that maximizes performance through data locality, Doris has embraced a modern compute-storage decoupled architecture tailored for cloud-native environments.[6] This elastic model separates stateless compute resources from a shared, low-cost storage layer, enabling independent scaling, fine-grained workload isolation, and significant cost efficiencies—critical requirements for contemporary data platforms.

The system's remarkable performance is not the result of a single feature but a synergy of sophisticated internal components. Its foundation is a columnar storage engine that minimizes

I/O and maximizes data compression, augmented by a hybrid row-columnar format to accelerate point queries.[2] Atop this storage layer, a state-of-the-art query execution engine combines the raw CPU efficiency of vectorization with the high-concurrency capabilities of a non-blocking Pipeline model.[1] This execution is guided by an advanced query optimizer that leverages both rule-based heuristics and cost-based analysis to craft highly efficient distributed plans.[10]

This report aims to deconstruct these core components and their intricate interactions. It will explore the physical and logical organization of data, the complete lifecycle of a query from SQL text to final result set, the various mechanisms for data ingestion and maintenance, and the advanced features like materialized views and multi-layered indexing that accelerate performance. By providing a comprehensive analysis of the design trade-offs and engineering principles embedded within Doris, this document serves as a definitive guide for data architects, principal engineers, and database researchers seeking to understand and leverage the full power of this next-generation analytical database.

Section 2: Core Architectural Principles

The architecture of Apache Doris is defined by a commitment to simplicity, scalability, and high availability. Its design revolves around two primary processes—Frontend (FE) and Backend (BE)—which collaborate to provide a unified and robust analytical platform. This foundational model has evolved to support both tightly-coupled and decoupled deployment patterns, offering flexibility to meet diverse performance and cost requirements.

2.1 The Two-Process Model: Frontend (FE) and Backend (BE)

Unlike many big data systems that rely on a complex stack of external components, Doris is built on a highly integrated, self-contained architecture consisting of only two core service types. This design choice is fundamental to its reputation for ease of deployment and low operational overhead.[2]

Frontend (FE) Deep Dive

The Frontend (FE) node serves as the brain and coordinator of the entire Doris cluster. Its responsibilities are multifaceted, encompassing client interaction, query management, and cluster governance.[6]

- **Responsibilities:** The FE is the entry point for all client requests. It is compatible with the MySQL protocol, a crucial feature that enables seamless integration with a vast ecosystem of existing BI tools, SQL clients, and data integration platforms.[2] Upon receiving an SQL query, the FE performs parsing, semantic analysis, query planning, and optimization to generate an efficient distributed execution plan.[1] It also manages all cluster-wide metadata, including table schemas, partitioning rules, data replica locations, and user access privileges.[6] Finally, it monitors the health of all Backend nodes and orchestrates query execution across the cluster.
- **High Availability (HA):** To prevent a single point of failure, FE nodes are deployed in a high-availability cluster. They use a Paxos-like consistency protocol, specifically Berkeley DB Java Edition (BDB JE), to replicate metadata changes and elect a leader (Master).[13] The cluster consists of three roles:
    - **Master:** A single, elected FE node responsible for all metadata write operations.
    - **Follower:** Replicates metadata from the Master and can serve metadata read requests. In the event of a Master failure, a Follower is elected as the new Master.
    - **Observer:** Replicates metadata and serves read requests but does not participate in leader elections. Observers are primarily used to scale out the cluster's capacity for handling concurrent queries without adding to the election overhead.[1]

      This multi-role setup ensures both metadata durability and high read throughput. The FE processes themselves are horizontally scalable, allowing more nodes to be added to handle increased client connections and query planning load.1
- **Metadata Management:** The FE stores all system metadata in its memory for fast access and persists it to a local disk, typically in a directory named doris-meta.[14] This comprehensive metadata catalog is the single source of truth for the entire cluster's state.

Backend (BE) Deep Dive

The Backend (BE) node is the workhorse of the Doris cluster, handling the heavy lifting of data storage and computation.[6]

- **Responsibilities:** The primary functions of a BE are to store data tablets (the physical shards of data) and to execute the query fragments dispatched by the FE.[1] When a query plan is executed, BEs scan local data, perform computations like filtering and aggregation, and shuffle intermediate data to other BEs as required by the plan. BEs also manage various background tasks essential for system health and performance, such as data compaction, schema changes, and replica balancing.[11]

- **Scalability and Data Reliability:** BEs are designed for horizontal scalability; new nodes can be added to the cluster to linearly increase both storage capacity and computational power, supporting clusters with hundreds of nodes and petabytes of data.[1] Data reliability is ensured by storing multiple replicas of each data tablet on different BE nodes. A quorum-based replication protocol is used for writes, ensuring that data is durable even if some nodes fail.[13] The system can automatically detect node failures and clone missing replicas to restore the desired replication level, providing self-healing capabilities.[12]

FE-BE Interaction

The FE and BE nodes maintain continuous communication to ensure smooth cluster operation:

- BEs send regular heartbeat signals to the FE, which contain information about node health, disk usage, and tablet status.[18]
- The FE dispatches query plan fragments to the appropriate BEs based on data locality information stored in its metadata.[9]
- BEs report the status of their tasks (e.g., query execution, data loading, compaction) back to the FE, which updates the global state accordingly.[19]
- Cluster topology changes, such as adding a new BE node via the ADD BACKEND command, are managed by the FE, which then orchestrates the process of data rebalancing to utilize the new resources.[18]

2.2 Architectural Evolution: From Coupled to Decoupled

Doris was initially conceived with a classic compute-storage coupled architecture but has since evolved to embrace a more flexible, cloud-native decoupled model. This evolution reflects a strategic response to the changing demands of modern data platforms, particularly the need for elasticity and cost optimization.

The Classic Coupled Architecture (Shared-Nothing)

The traditional deployment model for Doris is a Shared-Nothing architecture where compute and storage are tightly co-located on the same BE nodes.[6] This design is a hallmark of many high-performance MPP databases.

- **Design:** Each BE node manages its own local disks and processes queries on the data it stores. This maximizes data locality, as computations can often be performed without transferring data over the network, which is a primary source of latency in distributed systems.[6]
- **Trade-offs:** This model offers excellent performance and is relatively simple to deploy as it has no external system dependencies.[6] However, it presents a significant challenge: compute and storage resources are scaled together. If a workload requires more computational power but not more storage (or vice versa), one set of resources is inevitably over-provisioned, leading to inefficiency and higher costs.

The Modern Decoupled Architecture

To address the limitations of the coupled model, Doris introduced a compute-storage decoupled architecture, which is particularly well-suited for cloud environments.[6] This model fundamentally restructures the system into three distinct tiers.[7]

- **Three-Tier Structure:**
  1. **Shared Storage Layer:** Instead of being stored on local BE disks, the primary data files (segments and indexes) are persisted in a centralized, shared storage system like a cloud object store (e.g., Amazon S3) or HDFS.[6] These systems offer low-cost, highly durable storage, and their management is offloaded from Doris, simplifying operations.[7]
  2. **Stateless Compute Layer:** In this model, BE nodes become stateless compute resources. They do not store primary data permanently. To mitigate the performance penalty of accessing remote data, BEs use their local disks as a high-speed cache for frequently accessed data blocks.[6] This allows hot data to be served with performance close to that of the coupled model, while cold data resides in cheaper object storage. These stateless BEs can be organized into logical **Compute Groups**, allowing different workloads or teams to have their own dedicated, elastically scalable compute resources while operating on the same shared data.[6]
  3. **Meta Service:** A new, stateless service is introduced to manage metadata in the decoupled mode. It handles critical functions like data import transaction processing and tablet metadata management, and it can be scaled horizontally to meet demand.[6]

Analysis of Architectural Trade-offs

The availability of both architectures allows users to choose the model that best fits their specific needs. This choice is not merely a technical detail but a strategic decision that impacts performance, cost, and operational flexibility. The move toward a decoupled architecture represents a fundamental pivot in Doris's strategy, aiming to capture the benefits of cloud-native design. This shift transforms the system's economic model. For instance, in the coupled model, data compaction must run on all three replicas, consuming three times the compute resources. In the decoupled model, with a single data replica in object storage, compaction runs only once, drastically reducing computational waste and cost.[20] This positions Doris to compete effectively with leading cloud-native data warehouses by offering the elasticity, multi-tenancy, and cost-efficiency that modern enterprises demand.

- **Coupled Architecture:** This model remains the optimal choice for performance-critical applications where minimizing I/O latency is the absolute priority. Its self-contained nature and reliance on data locality deliver the highest possible query speeds, making it ideal for on-premise deployments or scenarios where predictable, low-latency performance outweighs the benefits of elastic scaling.[6]
- **Decoupled Architecture:** This model is the clear choice for cloud-native deployments and workloads with variable demand. It provides unparalleled **elasticity**, allowing compute resources to be scaled up or down independently of storage to match workload needs, thereby optimizing costs. It enables true **workload isolation** by allowing different teams to use separate Compute Groups on a shared data lake, preventing resource contention. Finally, it dramatically lowers **storage costs** by leveraging inexpensive object storage for the bulk of the data.[6]

Section 3: The Storage Engine: Physical and Logical Data Organization

The performance of any analytical database is fundamentally tied to how it stores and organizes data on disk. Apache Doris employs a highly optimized storage engine centered around a columnar format, complemented by sophisticated logical data models and distribution strategies. This multi-layered approach is designed to minimize I/O, maximize data compression, and enable massive parallelism during query execution.

3.1 Columnar Storage and Its Optimizations

The cornerstone of the Doris storage engine is its columnar storage format. Unlike traditional row-oriented databases that store all fields of a row contiguously, a columnar database stores all values for a single column together.[2]

- **Core Principle:** This design is exceptionally well-suited for Online Analytical Processing (OLAP) workloads. Analytical queries typically aggregate or filter data based on a small subset of columns across millions or billions of rows. With columnar storage, the query engine only needs to read the data for the columns referenced in the query, dramatically reducing the amount of data read from disk compared to a row-oriented system that would have to read entire rows.[13]
- **Segment v2 File Format:** Data on disk is physically organized into files called Segments. The "Segment v2" format is the modern structure used by Doris, which is meticulously designed for performance. Each Segment file is composed of three main parts [23]:
    1. **Data Region:** This section contains the actual column data. The data for each column is stored separately and is further divided into smaller blocks called Pages (typically 64 KB). This paged structure allows for fine-grained data access and caching.[23]
    2. **Index Region:** To avoid loading data unnecessarily, all index structures (such as Ordinal, ZoneMap, and Bloom Filter indexes) are stored together in a separate region. This allows the query engine to load and consult the indexes for an entire column without touching the data pages themselves, enabling efficient data pruning.[23]
    3. **Footer:** The end of the file contains a footer with critical metadata about the segment. This includes schema information for each column, pointers to the location of each column's data and index pages, and a short key index for rapid seeking.[23]
- **Encoding and Compression:** The homogeneity of data within a column makes it highly compressible. Doris applies various encoding and compression techniques tailored to the data type to reduce storage footprint and accelerate I/O. For example, it may use BIT_SHUFFLE for numeric types, Dictionary (DICT) encoding for low-cardinality strings, and Run-Length Encoding (RLE) for boolean values. These encodings not only save space but can also accelerate computations directly on the encoded data.[22]
- **Hybrid Row-Columnar Storage:** While pure columnar storage excels at analytical scans, it introduces a significant performance penalty for point queries that retrieve an entire row (e.g., SELECT * FROM table WHERE primary_key = 'some_id'). In a wide table with hundreds of columns, such a query would require hundreds of separate disk I/Os, creating an I/O bottleneck.[8] To address this, Doris offers an optional hybrid row-columnar storage model. When a user enables the
  "store_row_column" = "true" property on a table, Doris creates an additional, hidden

column. This special column stores a binary-concatenated version of all the other columns in the row.[8] When a point query is executed, the engine can retrieve the entire row's data with a single read from this row-store column, drastically improving performance for high-concurrency lookup scenarios.[26] This feature represents a pragmatic compromise, positioning Doris for workloads that blur the line between pure OLAP and Hybrid Transactional/Analytical Processing (HTAP). It acknowledges that many real-world applications, such as user-facing dashboards, require both fast aggregations and rapid retrieval of individual records. This flexibility comes at the cost of increased storage consumption, a trade-off that allows architects to optimize for a wider range of query patterns.[12]

3.2 Logical Data Models: Tailoring Storage to Workloads

Beyond the physical layout, Doris requires users to define a logical data model at the time of table creation. This choice is immutable and fundamentally dictates how data is handled during ingestion and storage, effectively acting as a "schema-on-write" optimization strategy.[27] This approach forces upfront design decisions to maximize query-time performance, contrasting with schema-on-read systems that offer more flexibility but often at the cost of slower query execution. Doris provides three distinct models [27]:

- **Duplicate Key Model:** This is the simplest model, storing all ingested rows exactly as they are, without any uniqueness or aggregation constraints.[27] It is ideal for storing raw, immutable fact data, such as logs or event streams. This model offers the highest ingestion throughput and maximum flexibility for ad-hoc queries across any combination of dimensions, as it is not constrained by a pre-defined aggregation logic.[13]
- **Aggregate Key Model:** This model is designed for reporting and dashboarding scenarios with fixed query patterns. During data ingestion, rows that share the same key columns are automatically aggregated. The non-key "value" columns are combined using a specified aggregation function (e.g., SUM, MIN, MAX, REPLACE). This pre-aggregation dramatically reduces the amount of data stored on disk and significantly accelerates queries that align with the aggregation definition.[27]
- **Unique Key Model:** This model enforces the uniqueness of the key columns, effectively implementing an "upsert" (update or insert) semantic. When new data arrives with a key that already exists, the old row is replaced with the new one. This model is essential for use cases that require synchronizing data from transactional systems (e.g., via Change Data Capture) or any scenario involving frequent updates to existing records.[27] To improve update performance, Doris offers a high-performance Merge-on-Write implementation for this model.[28]

3.3 Data Distribution: Partitioning and Bucketing

To manage massive datasets and enable parallel processing, Doris employs a two-level data distribution strategy: partitioning and bucketing.[29]

- **Two-Layer Sharding:** A table's data is first logically divided into **Partitions**. Each partition is then further physically divided into **Buckets**. This two-tiered approach provides a powerful and flexible way to manage and query data.[31]
- **Partitioning:**
  - **Purpose:** Partitioning is primarily used for data lifecycle management and query performance optimization through partition pruning. By dividing a table into smaller logical chunks based on a partition key, operations like deleting old data become trivial (simply drop a partition), and queries with filters on the partition key can skip reading irrelevant partitions entirely.[30]
  - **Types and Management:** Doris supports **Range Partitioning**, which is most common for time-series data (e.g., partitioning by day or month on a DATETIME column), and **List Partitioning**, for partitioning on discrete values (e.g., country code or city name).[30] Partitions can be created manually, or their creation and deletion can be automated using the
    Dynamic Partitioning feature, which is ideal for managing rolling time windows of data.[30]
- **Bucketing:**
  - **Purpose:** While partitioning provides logical data management, bucketing is about physical data distribution. Its goal is to evenly distribute the data within a partition across all BE nodes in the cluster to enable balanced, parallel query execution.[30] It is also the key mechanism that enables advanced query optimizations like Colocate Join.
  - **Methods:** The primary method is **Hash Bucketing**, where data is assigned to a bucket based on the hash value of one or more specified bucket key columns.[34] For cases where a suitable, evenly distributed hash key is not available,
    Random Bucketing can be used to prevent data skew by distributing rows randomly.[34]
  - **Tablet:** The fundamental unit of physical storage, replication, and balancing in Doris is the **Tablet**. A tablet is simply a bucket of data within a specific partition.[30] For high availability, each tablet has multiple replicas stored on different BE nodes. The total number of tablets in a table is the number of partitions multiplied by the number of buckets per partition.[31]

The following table provides a clear, at-a-glance reference for architects to choose the correct data model based on their specific use case requirements for data updates, query patterns, and performance.

Table 3.1: Comparison of Doris Data Models

Feature	Duplicate Key Model	Aggregate Key Model	Unique Key Model
Primary Use Case	Storing raw, detailed fact data (e.g., logs, events) for ad-hoc analysis.	Pre-aggregated reporting and dashboards with fixed query patterns (e.g., calculating daily SUMs, MINs, MAXs).	Scenarios requiring real-time data updates or synchronization with transactional systems (upserts).
Key Uniqueness	Not enforced. Duplicate key rows are allowed and stored.	Enforced. Keys are unique within the aggregated result.	Enforced. Keys are unique, ensuring only one row per primary key.
Data Ingestion Behavior	All incoming rows are appended as-is. Highest ingestion performance.	Rows with the same key are aggregated based on a defined function (SUM, REPLACE, etc.) during ingestion.	New rows with an existing key overwrite (update) the old row.
Update/Delete Support	No support for UPDATE or DELETE statements.	No support for UPDATE or DELETE. Partial column updates possible via REPLACE_IF_NOT_NULL aggregation.	Full support for UPDATE and DELETE statements. Supports partial column updates.
Query Performance	Excellent for ad-hoc queries across any dimension. No	Extremely fast for queries that match the pre-aggregation.	Good for point queries and updates. Cannot leverage

	pre-aggregation benefits.	Less flexible for other query patterns.	pre-aggregation benefits for analytical queries.

Section 4: Query Lifecycle: From SQL to Results

The journey of a query in Apache Doris is a sophisticated, multi-stage process designed to transform a declarative SQL statement into a highly optimized, parallelized execution plan. This process is orchestrated by the Frontend (FE) node and executed by the Backend (BE) nodes, leveraging an advanced query optimizer and a modern execution engine to achieve high performance.

4.1 The Query Optimizer: Crafting the Optimal Plan

The query optimizer is the intelligent core of the FE, responsible for finding the most efficient way to execute a given query. It employs a combination of rule-based and cost-based techniques to explore a vast search space of possible execution plans and select the best one.[10]

- **Multi-Stage Process:** The optimization pipeline begins after a client sends an SQL string to the FE.
  1. **Syntax and Semantic Analysis:** The SQL text is first parsed into an Abstract Syntax Tree (AST). The analyzer then validates this tree, checking for the existence of tables and columns, verifying data types, and resolving function calls. If any errors are found, the query is rejected.[10]
  2. **Rule-Based Optimization (RBO):** The valid AST is transformed into an initial logical query plan. The RBO phase then applies a series of deterministic, heuristic-based rules to rewrite this plan into a more efficient form. Key transformations include:
     - **Predicate Pushdown:** Moving WHERE clause filters as close to the data source as possible to reduce the volume of data processed in later stages.[2]
     - **Partition Pruning:** Eliminating entire data partitions from the scan if they cannot possibly contain data that satisfies the query's filters.
     - **Constant Folding:** Evaluating constant expressions (e.g., 1 + 1) at planning time rather than execution time.
     - **Subquery Rewriting:** Transforming complex subqueries into more efficient join

operations.[10]

3. **Cost-Based Optimization (CBO):** Following RBO, the CBO takes over to handle more complex decisions, particularly the optimal join order for multi-table queries. Doris's CBO is based on the advanced Cascades optimization framework.[10] It systematically explores various equivalent logical plans (e.g., joining tables A and B first, then C, versus joining B and C first, then A). For each potential plan, it uses stored statistics about the data (such as table cardinality and column value distribution) to estimate its execution "cost" in terms of CPU, I/O, and network resources. The plan with the lowest estimated cost is ultimately chosen for execution.[2]

- **Adaptive Query Execution with Runtime Filters:** Doris's optimization does not stop at the planning phase; it continues dynamically during query execution. The most powerful adaptive technique is the **Runtime Filter**. When executing a hash join, the build side (typically the smaller table) constructs a filter based on the actual join key values it processes. This filter, which can be a Min/Max range, an IN predicate, or a Bloom Filter, is then sent to the BE nodes that are scanning the probe side (the larger table). The scan operator on the probe side applies this filter *before* reading data from disk, allowing it to skip entire data blocks or pages that could not possibly match the join condition. This technique can dramatically reduce I/O and network traffic, leading to significant performance gains, especially in selective joins.[1]

4.2 The Execution Engine: From Volcano to Pipeline

The execution engine, residing on the BE nodes, is responsible for bringing the physical plan to life. Doris has transitioned from a traditional execution model to a modern, highly parallel one.

- **Legacy Volcano Model:** Early versions of Doris used the classic "Volcano" iterator model. In this pull-based system, each operator in the plan tree has a next() method. To get results, a parent operator calls next() on its child, which in turn calls next() on its own child, and so on, pulling data up the tree one row or batch at a time. While simple to implement, this model can suffer from inefficiencies, as a blocked operator (e.g., waiting for network data) can stall an entire execution thread.[9]
- **Modern Pipeline Execution Engine:** To overcome the limitations of the Volcano model and handle high concurrency effectively, Doris implemented a Pipeline execution engine. This model is a direct and sophisticated solution to the classic MPP concurrency bottleneck where assigning one thread per query fragment leads to a "thread explosion" under high load, causing excessive context switching and overhead.[1]
  - **Core Concept:** The execution plan is broken down into a series of **pipelines**. A pipeline is a sequence of operators that can execute without blocking. Operators that

inherently block, such as the build phase of a hash join or a network exchange, act as "pipeline breakers." They split the plan into multiple, dependent pipelines.[9]
    - **Benefits:** Instead of dedicating a thread to each part of a query, the Pipeline engine schedules executable pipeline tasks onto a fixed-size thread pool (usually sized to the number of available CPU cores). When a task needs to block (e.g., to wait for I/O), it yields its thread, which the scheduler can immediately assign to another ready task. This non-blocking, task-based scheduling model prevents threads from being held hostage by slow operations, dramatically improving overall CPU utilization and system throughput under high concurrency.[9]
- **Fully Vectorized Engine:** Complementing the Pipeline model is a fully vectorized execution engine.
    - **Core Concept:** The engine processes data not row-by-row, but in batches (typically of 1024 rows) called vectors or blocks. These blocks are laid out in a columnar format in memory, mirroring the on-disk storage format.[2]
    - **Benefits:** This approach yields massive performance improvements by:
        1. **Reducing Per-Row Overhead:** The cost of function calls and control flow logic is amortized over an entire batch of rows.
        2. **Improving CPU Cache Locality:** Processing a single column's data sequentially leads to better cache utilization.
        3. Enabling SIMD: It allows the engine to leverage modern CPU's SIMD (Single Instruction, Multiple Data) instructions, which can perform the same operation (e.g., addition, comparison) on multiple data points simultaneously within a single CPU cycle.
           This combination of optimizations results in a 5 to 10-fold performance increase in CPU-bound analytical workloads like wide table aggregations.4

The synergy between the CBO, Runtime Filters, and the Vectorized Engine creates a multi-layered performance strategy. The CBO provides static, global planning before execution begins. The Vectorized Engine optimizes the micro-level execution of each operation for maximum CPU efficiency. Finally, Runtime Filters add a dynamic, adaptive layer that optimizes data flow between operators at runtime. This comprehensive approach makes the query engine robust and performant across a wide variety of query patterns and data distributions.

4.3 Distributed Execution Plan

The final step in the FE is to convert the optimized single-node plan into a distributed plan that can be executed in parallel across the BE cluster.

- **From Logical to Physical Plan:** The optimizer inserts ExchangeNode operators into the plan. These nodes represent data transfer (shuffling) points between BEs. For example, in a hash join between two tables distributed across the cluster, an ExchangeNode is needed to re-distribute the data from both tables to the BEs based on the hash of the join key, ensuring that rows with the same key end up on the same node for joining.[39]

- **Fragments:** The physical plan is then partitioned into **Fragments**. A Fragment is a self-contained sub-plan that can be executed on a single BE node without interruption. The boundaries of fragments are typically the ExchangeNodes.[9] The FE dispatches each fragment to one or more BEs for execution.
- **PipelineTasks:** On a BE, each received Fragment is further compiled into a set of logical Pipelines. Each logical Pipeline is then instantiated into multiple parallel **PipelineTasks**, which are the actual units of work scheduled onto the engine's thread pool. This allows for both inter-node parallelism (multiple BEs working on different fragments) and intra-node parallelism (multiple threads on one BE working on different PipelineTasks).[39]
- **Local Shuffle:** To prevent data skew *within* a single BE node, where one PipelineTask might receive significantly more data than others, the planner can insert a Local Shuffle operator. This operator acts as a pipeline breaker that re-distributes the data from an upstream pipeline evenly among all the tasks of a downstream pipeline, ensuring balanced execution and maximizing intra-node parallelism.[9]

Section 5: Data Management and System Operations

Effective data management is crucial for the performance and reliability of a data warehouse. Apache Doris provides a robust set of tools and automated processes for data ingestion, storage maintenance, and ensuring data consistency and concurrency. These internal operations are designed to support the demands of real-time analytics, from high-frequency data streams to large-scale batch updates.

5.1 Data Ingestion Mechanisms

Doris's role as a centralized analytical hub is reflected in its diverse and flexible data ingestion capabilities. It supports a comprehensive suite of methods catering to different data sources, volumes, and latency requirements, allowing it to integrate seamlessly into virtually any data architecture.[29]

- **Stream Load:** This is a synchronous, push-based method where data is sent to a BE node via an HTTP request. The client application reads the data (e.g., from a local file or a data stream) and pushes it directly to Doris. The HTTP response immediately indicates the success or failure of the load job, making it suitable for scenarios that require real-time feedback and low-latency ingestion.[12]

- **Broker Load:** This is an asynchronous, pull-based method designed for large-scale batch loading from remote storage systems like HDFS or cloud object stores (S3, GCS, etc.).[29] The user submits a
  LOAD command to the FE, which then orchestrates the import. The BE nodes pull the data directly from the remote source in parallel. This method is ideal for offline ETL jobs and initial data migrations, as it offloads the data transfer work from the client to the Doris cluster itself.[46]
- **Routine Load:** This method provides continuous, managed data ingestion from Apache Kafka. A user creates a persistent ROUTINE LOAD job on the FE, specifying the Kafka topic and connection details. The FE then automatically and continuously creates small, transactional micro-batch tasks to consume messages from Kafka and load them into Doris. This approach is designed for robust, real-time streaming pipelines and supports Exactly-Once semantics to prevent data loss or duplication.[12]
- **Insert Into:** Doris supports the standard SQL INSERT INTO statement, which serves two primary purposes. INSERT INTO... VALUES is a synchronous command suitable for inserting small amounts of data for testing or ad-hoc purposes. More powerfully, INSERT INTO... SELECT acts as an internal ETL tool, allowing users to load data into a Doris table from the result of any valid SELECT query. This can be used to transform and move data between Doris tables or to ingest data from external tables defined via the Multi-Catalog feature.[12]
- **Spark Load:** For extremely large datasets that require significant preprocessing before ingestion, Doris offers Spark Load. This method leverages an external Apache Spark cluster to perform complex transformations, sorting, and pre-aggregation on the source data. The processed data is then efficiently loaded into Doris. This offloads the resource-intensive ETL work from the BEs, preserving their resources for query execution.[12]

5.2 Data Compaction and Maintenance

Doris employs a storage architecture with characteristics similar to a Log-Structured Merge-Tree (LSM-Tree). When data is written, it creates new, immutable segment files on disk. This append-only approach is highly efficient for writes, but over time it can lead to a proliferation of small files, which would degrade read performance. **Compaction** is the critical background process that mitigates this issue.[11]

- **Purpose of Compaction:** The primary goal of compaction is to periodically merge smaller segment files into larger, more optimally structured ones. This process improves query performance by reducing the number of files that need to be opened and scanned. Critically, compaction is also the mechanism by which data updates and deletions (in the Unique Key model) are physically applied to the data files.[16]

- **Compaction Strategies:** Doris has evolved its compaction algorithms to be more efficient and less resource-intensive.
  - **Cumulative and Base Compaction:** The system distinguishes between two main types of compaction. **Cumulative Compaction** focuses on merging newly ingested, smaller, and more numerous rowsets quickly. **Base Compaction** is a heavier process that merges larger, older rowsets to create the final, optimized base version of the data.[51]
  - **Vertical Compaction:** For wide tables (tables with many columns), traditional compaction can be extremely memory-intensive as it needs to load all columns for the rows being merged. Vertical Compaction is an optimized algorithm that processes columns in smaller, independent groups. This dramatically reduces the peak memory usage (by up to 90%) and increases the speed of compaction, making updates and maintenance on wide tables far more efficient and stable.[16]
  - **Segment Compaction:** To prevent load jobs from failing with a "too many segments" error due to high-frequency writes or low-cardinality data, this feature can merge small segment files *as they are being generated* during the load process itself. This ensures that the base data remains healthy and improves the performance of subsequent queries on the newly loaded data.[16]
  - **Single Replica Compaction:** In a standard multi-replica setup, each replica would independently perform compaction, consuming CPU and I/O resources on each node. To optimize this, the Single Replica Compaction strategy designates one replica to perform the compaction work. Once complete, the other replicas simply download the already compacted files. This approach saves N-1 times the computational resources, where N is the number of replicas, significantly reducing the background load on the cluster.[16]

5.3 Concurrency, Consistency, and Transactions

Doris provides strong guarantees around data consistency and manages concurrent operations to ensure system stability and predictable behavior.

- **Transactionality and Atomicity:** Every data import job in Doris, regardless of the method used, is treated as a single atomic transaction. This ensures that all data within a given load batch is either successfully committed and made visible, or the entire batch is rolled back in case of failure. There is no risk of partial data writes.[52]
- **Multi-Version Concurrency Control (MVCC):** Doris uses an MVCC mechanism to

handle concurrent reads and writes. Each committed transaction is assigned a unique, monotonically increasing version number. When a query is executed, it is given a specific version number to read. This allows the query to see a consistent snapshot of the data as of that version, without being affected by concurrent write operations. Similarly, writers do not block readers, enabling high concurrency for mixed workloads.[54]

- **Consistency Model:**
  - **Write Consistency:** For data durability, Doris relies on a multi-replica architecture. By default, it employs a **majority write** (quorum) strategy. For a table with three replicas, a write transaction is considered successful and is committed only after at least two of the three replicas have successfully written the data. The remaining replica is then updated asynchronously in the background. This model provides a robust balance between data reliability and write latency.[13]
  - **Data Updates:** In the Unique Key model, MVCC versioning is used to resolve update conflicts. When multiple updates for the same primary key occur, the data from the transaction with the highest version number prevails. To handle cases where data may arrive out of order from parallel sources, users can define a sequence column in their table. Doris will then use the value in this column as a tie-breaker to determine the correct update order, ensuring that older data does not overwrite newer data.[12]
- **Concurrency Control:** Doris provides mechanisms to manage both query and write concurrency to prevent system overload.
  - **Query Concurrency:** This is managed through **Workload Groups**. Administrators can define groups with specific limits on max_concurrency (the number of queries running at once), max_queue_size (how many queries can wait), and queue_timeout. When the concurrency limit is reached, new queries are queued or rejected, ensuring the system remains stable under heavy load.[55]
  - **Write Concurrency:** High-frequency, small-batch writes (e.g., from streaming sources) can create significant overhead for the FE, which has to manage a transaction for each write. The **Group Commit** mechanism addresses this by batching multiple small, independent write requests on the server-side into a single, larger transaction. This dramatically reduces the load on the FE and limits the rapid proliferation of small file versions on the BEs, improving overall system throughput and stability.[56] The combination of advanced compaction algorithms and Group Commit is what makes real-time, high-frequency updates a viable and performant feature in Doris, distinguishing it from more traditional, append-only OLAP systems.

The following table provides a clear framework for users to select the appropriate data loading method by outlining the characteristics, use cases, and operational model of each.

Table 5.1: Comparison of Primary Data Ingestion Methods

Feature	Stream Load	Broker Load	Routine Load	Insert Into

Mechanism	Push-based (Client pushes data to BE)	Pull-based (BE pulls data from remote source)	Pull-based (FE manages continuous pull from Kafka)	SQL-based (Executed within the Doris cluster)
Synchronicity	Synchronous (Immediate success/fail response)	Asynchronous (Job status checked via SHOW LOAD)	Asynchronous (Persistent job, status checked via SHOW ROUTINE LOAD)	Synchronous (Immediate success/fail response)
Typical Data Source	Local files, application data streams via HTTP	HDFS, S3, and other object stores	Apache Kafka topics	Other Doris tables, results of a SELECT query, or literal VALUES
Recommended Data Volume	Small to medium batches (MBs to a few GBs) per request	Large batches (tens to hundreds of GBs) per job	Continuous micro-batches (KBs to MBs)	Small for VALUES, variable for SELECT (depends on query result size)
Key Use Case	Real-time streaming ingestion from applications; loading local files.	Large-scale, offline batch ETL from data lakes or object storage.	Building robust, continuous, real-time streaming pipelines from Kafka.	Internal data transformation (ETL); loading from external tables; small ad-hoc inserts.

Section 6: Performance Acceleration and Advanced Features

Beyond its core architecture, Apache Doris incorporates a suite of advanced features designed specifically to accelerate query performance and reduce computational overhead. These mechanisms, including materialized views, a multi-layered indexing strategy, and intelligent caching, work in concert to deliver the sub-second response times required for modern analytical applications.

6.1 Materialized Views: Pre-computation for Speed

A materialized view (MV) is a powerful optimization technique that pre-computes and stores the result set of a query, allowing subsequent queries to access the pre-calculated data instead of re-computing it from the base tables. This trade of storage space for query speed is a cornerstone of performance tuning in data warehousing.[58]

- **Core Concept:** When a user submits a query, the Doris optimizer can automatically and transparently rewrite it to use a suitable materialized view, even if the query references the base tables directly. This process is seamless to the end-user, who continues to query the original tables but experiences significantly faster response times.[2] Doris supports two distinct types of materialized views, each representing a different trade-off between data freshness and computational complexity.
- **Synchronous Materialized Views:**
    - **Mechanism:** Also known historically as a Rollup, a synchronous MV is updated in real-time, within the same transaction as the base table. When new data is loaded, the corresponding aggregations in the MV are calculated and committed simultaneously with the base table data. This ensures **strong consistency** between the MV and its source.[59]
    - **Limitations:** This real-time consistency comes with constraints. Synchronous MVs can only be defined on a **single table** and cannot contain complex operations like joins. They are essentially pre-aggregated summaries of a single table. Furthermore, because they add computational overhead to every write operation, having too many synchronous MVs on a single table can slow down data ingestion performance.[58]
- **Asynchronous Materialized Views:**
    - **Mechanism:** Asynchronous MVs decouple the refresh process from the data ingestion pipeline. They are updated based on a defined schedule (e.g., every hour), a manual trigger, or upon a commit to the base table. This results in **eventual consistency**, where the data in the MV may lag slightly behind the base table.[61]
    - **Capabilities:** The flexibility gained from decoupling is immense. Asynchronous MVs are far more powerful, supporting definitions that include **multi-table joins**, complex expressions, and even queries on external data sources in a Lakehouse architecture.

They can also be queried directly as if they were regular tables, making them useful for creating intermediate data marts or summary tables in an ETL workflow.[59]

The existence of these two MV types is a deliberate design choice, offering architects a clear trade-off. Synchronous MVs are ideal for real-time dashboards and reporting on a single fact table where data freshness is paramount. Asynchronous MVs are better suited for building complex, cross-domain data marts or accelerating heavy analytical queries where a small amount of data latency is acceptable in exchange for the ability to pre-compute complex joins and transformations.

6.2 Multi-Layered Indexing Strategy

Doris employs a hierarchical indexing strategy that combines several types of indexes to prune data at different granularities, from entire files down to individual data blocks. This "defense-in-depth" approach to I/O reduction is critical for achieving fast query performance by minimizing the amount of data that must be read from disk.[63]

- **Built-in Indexes (Smart Indexes):** These indexes are created and maintained automatically by Doris without user intervention.
  - **Sorted Compound Key (Prefix) Index:** When a table is created, its data is physically sorted on disk according to the specified key columns (Duplicate, Unique, or Aggregate Key). Doris automatically creates a sparse index on the first 36 bytes of this sort key. This index stores an entry for every 1024 rows, pointing to the start of that data block. When a query includes a filter on the prefix of the sort key, this index allows the scanner to quickly seek to the relevant data blocks, bypassing the vast majority of the table's data.[1]
  - **ZoneMap Index:** Doris automatically creates a ZoneMap index for every column in every segment file. This index stores metadata for each data page, including the minimum and maximum values. When a query contains a predicate (e.g., WHERE sales > 500 AND sales < 1000), the engine first consults the ZoneMap. If the query's range does not overlap with a page's min/max range, that entire page can be skipped without being read from disk. This provides coarse-grained but effective data pruning for all columns.[23]
- **User-Created Secondary Indexes:** For more targeted performance optimization, users can create secondary indexes on specific columns.
  - **Inverted Index:** This index creates a mapping from a value (or a token, in the case of text) to the list of row IDs that contain it. It is exceptionally powerful for accelerating full-text search queries (using MATCH) and high-selectivity equality or range filters on high-cardinality columns. It essentially allows for direct lookups of qualifying rows, avoiding a full table scan.[1]

   - **Bloom Filter Index:** A Bloom filter is a space-efficient, probabilistic data structure that can quickly determine if an element is *definitely not* in a set. Doris can create a Bloom filter for each data block on a specified column. When an equality query (= or IN) is executed, the engine checks the Bloom filter first. If the filter indicates the value is not present in the block, the block is skipped. This is highly effective for filtering on high-cardinality columns like user_id or order_id.[1]
   - **Bitmap Index:** Best suited for columns with low to medium cardinality (e.g., gender, country, city), a bitmap index creates a separate bitmap for each distinct value in the column. These bitmaps can be combined with extremely fast bitwise operations (AND, OR) to resolve complex, multi-column predicates efficiently.[66]

6.3 Caching Mechanisms

To reduce latency, especially when accessing remote data or re-executing common queries, Doris implements several layers of caching.

- **Metadata Cache:** In a Data Lakehouse setup, repeatedly fetching metadata (schemas, partitions, file locations) from external sources like a Hive Metastore can be a significant performance bottleneck. Doris maintains a cache of this external metadata within the FE, with configurable expiration and refresh policies. This allows for millisecond-level metadata access for subsequent queries.[12]
- **Data Cache (Block Cache):** This cache is essential for the compute-storage decoupled architecture. When a BE node reads a data block from a remote object store, it can cache that block on its fast local SSD. Subsequent queries that need the same data block can then read it from the local cache, avoiding the high latency of a network round trip to the remote storage. This mechanism effectively keeps "hot" data close to the compute resources.[12]
- **Query Cache (SQL Cache):** Doris can cache the final result set of queries. If an identical SQL query is submitted again and the underlying data in the queried tables has not changed, Doris can serve the result directly from the cache, bypassing query planning and execution entirely. This provides the fastest possible response time for frequently executed, deterministic queries, such as those powering popular dashboard widgets.[69]

The following table helps developers and DBAs understand which index to use for a given query pattern and data characteristic.

Table 6.1: Overview of Indexing Mechanisms in Apache Doris

Index Type	Category	Primary	Supported	Supported	Key

		Use Case	Data Cardinality	Query Operators	Trade-off
Sorted Compound Key	Built-in	Accelerating range scans and equality lookups on the prefix of the sort key.	All	=, IN, >, <, BETWEEN on key prefix	Only one per table; effectiveness depends on query aligning with the sort order.
ZoneMap Index	Built-in	Coarse-grained data skipping for all columns based on min/max values.	All	=, IN, >, <, BETWEEN, IS NULL	Less effective for non-selective queries or columns with wide value ranges.
Inverted Index	Secondary	Full-text search and fast point/range queries on any column.	High	MATCH, =, IN, >, <, array_contains	Higher storage overhead and ingestion cost.
Bloom Filter Index	Secondary	Accelerating equality lookups on high-cardinality columns (e.g., IDs).	High	=, IN	Probabilistic (small chance of false positives); not useful for range queries.
Bitmap Index	Secondary	Accelerating queries with multiple	Low to Medium	=, IN	Inefficient for high-cardinality

		AND/OR conditions on low-cardinality columns.			columns due to large bitmap size.

Section 7: Comparative Architectural Analysis

To fully appreciate the internal design of Apache Doris, it is instructive to compare it with other prominent systems in the analytical database landscape. This analysis focuses on architectural philosophies and key design trade-offs against ClickHouse, StarRocks, and Trino, revealing how Doris positions itself in the market.

7.1 Doris vs. ClickHouse

The comparison between Doris and ClickHouse highlights a fundamental difference in design philosophy: managed usability versus raw, specialized performance.

- **Architecture:** Doris is built around a managed cluster architecture with its FE/BE model. The FE provides a centralized point of control, automating tasks like data balancing, replica repair, and cluster expansion, which simplifies operations significantly.[71] ClickHouse, while capable of running in a cluster, has a design more akin to a collection of powerful single nodes. Setting up a distributed ClickHouse cluster requires manual configuration of distributed tables and relies on an external coordination service like ZooKeeper or ClickHouse Keeper, placing a higher operational burden on the user.[71]
- **Data Updates:** This is a key differentiator. Doris, with its **Unique Key model**, provides strongly consistent, synchronous update and delete operations. When an UPDATE command completes, the data is immediately consistent and visible to subsequent queries. This is critical for real-time analytics scenarios that require synchronization with transactional data.[72] ClickHouse's update and delete operations (via ALTER TABLE... UPDATE/DELETE) are asynchronous mutations. The operation is queued and executed in the background, meaning there can be a delay before the changes are reflected in query results, leading to temporary data inconsistencies.[71]
- **Joins and SQL Compatibility:** Doris was designed from the ground up as an MPP

system with a sophisticated CBO capable of handling complex, large-scale multi-table joins, including distributed shuffle joins between large tables.[71] ClickHouse is renowned for its world-class performance on single-table scans and queries on denormalized (wide) tables but has historically been less optimized for the kind of complex, ad-hoc joins common in BI tools.[73] Furthermore, Doris prioritizes compatibility with the MySQL protocol and standard ANSI SQL, which lowers the learning curve and simplifies integration with the existing data ecosystem. ClickHouse uses its own powerful but distinct SQL dialect, which may require users to adapt their queries and tools.[71]

7.2 Doris vs. StarRocks

Doris and StarRocks share a common ancestry, as StarRocks was forked from an early version of Apache Doris in 2020.[74] While they retain a similar foundational FE/BE architecture, both projects have undergone extensive, independent development and have diverged significantly in their focus and implementation.

- **Origins and Divergence:** Since the fork, both codebases have been heavily rewritten.[74] This shared heritage means they have similar concepts (e.g., data models, load methods), but the underlying implementation of core components like the query optimizer and execution engine has evolved separately.
- **Strategic Focus:** The two projects appear to be pursuing different strategic priorities. StarRocks has heavily marketed its capabilities as a high-performance query engine for data lakes, positioning itself as a direct competitor to and replacement for engines like Trino.[74] While Apache Doris also possesses strong data lake query capabilities, its development roadmap has maintained a broader focus on being a **unified OLAP database**. This includes significant investment in features that enhance its role as a standalone data warehouse, such as advanced support for semi-structured data (JSON, VARIANT types), optimizations for high-concurrency point queries, and robust real-time data update mechanisms.[74]
- **Performance and Community:** Both systems are highly performant, leveraging vectorized execution engines and cost-based optimizers. Publicly available benchmarks are often contentious and workload-dependent, with each project claiming performance leadership in different scenarios.[74] The choice between them often comes down to specific feature maturity, community support, and alignment with an organization's primary use case (e.g., pure lakehouse query vs. a unified data warehouse).

7.3 Doris vs. Trino (Presto)

The comparison with Trino highlights the difference between a unified data warehouse and a federated query engine.

- **Architecture:** This is the most fundamental distinction. Doris is an integrated system that combines both a storage engine and a query execution engine. It manages its own data in an optimized columnar format.[69] Trino, in contrast, is a pure **federated query engine**. It does *not* have its own storage layer. Its purpose is to execute queries on data where it currently resides, whether in a data lake (Hive, Iceberg, Hudi), a NoSQL database (Cassandra), or a relational database (MySQL, PostgreSQL).[69]
- **Performance:**
  - **On External Data:** When both are used as query engines on top of a data lake, Doris often demonstrates superior performance. This is attributed to its C++ implementation (versus Trino's Java), which allows for more direct memory management and CPU optimization, as well as more advanced caching mechanisms for both data and metadata, and the ability to use materialized views to accelerate queries on external tables.[69]
  - **On Native Data:** When data is ingested into Doris's own optimized storage format, the performance advantage becomes even more pronounced (often cited as 3-10x faster). This is because Doris can leverage its full suite of internal optimizations—columnar storage with compression, multi-layered indexing, data locality in the coupled model, and colocation strategies—which are not available to a storage-agnostic engine like Trino.[69]
- **Use Case and Philosophy:** Trino's strength lies in its ability to provide a single SQL interface for ad-hoc, exploratory queries across a vast and heterogeneous data landscape without requiring data movement. Doris offers a more **unified** solution. It can perform federated queries like Trino, but it also provides a high-performance native storage layer. This allows organizations to simplify their data architecture by using a single system for both data warehousing and data lake querying, eliminating the complexity and cost of maintaining separate storage and query engine layers.[69]

Ultimately, the competitive landscape reveals a clear "middle path" strategy for Doris. It does not aim to be the absolute fastest single-table engine, a title often claimed by ClickHouse, nor is it a pure, storage-agnostic query engine like Trino. Instead, Doris occupies a strategic middle ground, offering a balanced and unified platform. It provides better multi-table join performance, stronger data consistency, and simpler cluster management than ClickHouse, making it more suitable for general-purpose BI and enterprise analytics. Simultaneously, it delivers a more performant and integrated solution than Trino by tightly coupling storage and compute, providing a "one-stop shop" for a wide range of analytical needs.

The following table provides a high-level strategic comparison for architects choosing a system based on core design principles rather than just feature lists.

Table 7.1: Architectural Design Trade-offs: Doris vs. ClickHouse vs. Trino

Architectural Aspect	Apache Doris	ClickHouse	Trino (Presto)
Core Paradigm	**Unified Data Warehouse:** Integrates storage and compute for a comprehensive OLAP solution.	**High-Performance Columnar Database:** Optimized for extreme speed on single-table analytical queries.	**Federated Query Engine:** Decouples query execution from storage, querying data in-place.
Storage Model	Native, optimized columnar storage. Also supports querying external data lakes (Hive, Iceberg).	Native, highly optimized columnar storage via MergeTree engine family.	**None.** Relies entirely on external storage systems via connectors.
Data Update Model	**Synchronous & Consistent:** Unique Key model provides immediate, strongly consistent updates/deletes.	**Asynchronous & Eventual:** Mutations are processed in the background, leading to eventual consistency.	**Read-Only (by default):** Update capabilities depend entirely on the underlying storage system's connector.
Cluster Management	**Managed & Automated:** FE nodes handle cluster coordination, auto-balancing, and fault recovery.	**Manual & Flexible:** Requires external coordination (e.g., ZooKeeper) and manual setup of distributed tables.	**Coordinator/Worker Model:** Simple to scale workers, but no built-in data management or balancing.
Primary Strength	**Balance and Versatility:** Strong performance on complex joins, real-time updates,	**Raw Speed:** Unmatched performance on large, denormalized single-table scans	**Federation and Flexibility:** Ability to query anything, anywhere with a single SQL

	and ease of use.	and aggregations.	interface.

Section 8: Conclusion and Future Outlook

The internal architecture of Apache Doris is a testament to a pragmatic and evolving design philosophy, meticulously engineered to balance extreme performance with operational simplicity and data consistency. This analysis has deconstructed the key components that enable Doris to fulfill its role as a high-performance, real-time analytical data warehouse. The synergy between its core architectural tenets—the straightforward two-process FE/BE model, the flexible coupled and decoupled deployment options, a highly optimized columnar storage engine, and a modern query execution pipeline—creates a system that is both powerful and accessible.

The simplified FE/BE architecture significantly lowers the barrier to entry and reduces the total cost of ownership, a deliberate choice that contrasts sharply with the complexity of many distributed data systems. The evolution to include a compute-storage decoupled model is a strategic pivot that aligns Doris with the demands of cloud-native environments, offering the elasticity, workload isolation, and cost-efficiency required by modern enterprises. Internally, the storage engine's columnar format, augmented by a hybrid row-store option and a multi-layered indexing strategy, provides a versatile foundation for a wide spectrum of analytical and point-query workloads. This physical storage is intelligently managed through logical data models that allow architects to tailor data handling to specific business requirements for updates, aggregation, or raw data retention.

At the heart of its performance lies the query processing pipeline. The journey from SQL to result set is guided by a sophisticated, multi-stage optimizer that combines deterministic rule-based transformations with advanced cost-based planning. The resulting plan is brought to life by a state-of-the-art execution engine that leverages the CPU-level efficiency of vectorization and the high-concurrency, resource-efficient scheduling of the Pipeline model. This combination ensures that Doris can handle both high-throughput complex analysis and high-concurrency point queries effectively.

Ultimately, the design of Apache Doris reflects a deep understanding of the practical needs of enterprise data analytics. It prioritizes strong data consistency through transactional ingestion and synchronous update models, offers a familiar MySQL-compatible interface to foster broad ecosystem integration, and provides a unified platform that can simplify complex data architectures by serving as both a data warehouse and a data lake query engine.

Looking ahead, the project's trajectory indicates a continued commitment to these core principles. The ongoing development of the next-generation Nereids query optimizer promises even smarter and more robust query planning.[10] Further enhancements to its Data Lakehouse capabilities, including deeper integration with formats like Apache Iceberg and Apache Hudi and support for incremental materialized views on external data, will solidify its position as a central hub for unified analytics.[68] As Doris continues to evolve, its focus on delivering a balanced, powerful, and user-friendly analytical database ensures it will remain a compelling choice for organizations seeking to derive real-time insights from their data.

Works cited

1. Introduction to Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/gettingStarted/what-is-apache-doris
2. Introduction to Apache Doris - Medium, accessed August 19, 2025, https://medium.com/@ApacheDoris/introduction-to-apache-doris-incubating-48aaa1df34ae
3. Apache Doris is an easy-to-use, high performance and unified analytics database. - GitHub, accessed August 19, 2025, https://github.com/apache/doris
4. Introduction to Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/dev/gettingStarted/what-is-apache-doris/
5. How does Apache Doris help AISPEECH build a data warehouse in AI chatbots scenario, accessed August 19, 2025, https://doris.apache.org/blog/Use-Apache-Doris-with-AI-chatbots/
6. Overview - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/dev/compute-storage-decoupled/overview
7. Slash your cost by 90% with Apache Doris Compute-Storage Decoupled Mode, accessed August 19, 2025, https://doris.apache.org/blog/doris-compute-storage-decoupled
8. Hybrid Row-Columnar Storage - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/dev/table-design/row-store/
9. Steps to industry-leading query speed: evolution of the Apache Doris execution engine, accessed August 19, 2025, https://doris.apache.org/blog/evolution-of-the-apache-doris-execution-engine/
10. Query Optimizers - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/optimization-technology-principle/query-optimizer/
11. Introduction to Apache Doris Baeldung on SQL, accessed August 19, 2025, https://www.baeldung.com/sql/apache-doris-tutorial
12. Introduction to Apache Doris: a next-generation real-time data warehouse, accessed August 19, 2025, https://doris.apache.org/blog/introduction-to-apache-doris-a-next-generation-real-time-data-warehouse/
13. Introduction to Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/gettingStarted/what-is-apache-doris/
14. FE Configuration - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/admin-manual/config/fe-config

15. Deploying Apache Doris with sample Docker Compose results in FE and BE errors, accessed August 19, 2025, https://stackoverflow.com/questions/77572877/deploying-apache-doris-with-sample-docker-compose-results-in-fe-and-be-errors
16. Compaction - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/admin-manual/compaction/
17. Introduction to Apache Doris: A next-generation real-time data warehouse | VeloDB, accessed August 19, 2025, https://www.velodb.io/blog/127
18. ADD BACKEND - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/sql-manual/sql-statements/cluster-management/instance-management/ADD-BACKEND
19. Routine Load - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/data-operate/import/import-way/routine-load-manual/
20. Slash your cost by 90% with Apache Doris Compute-Storage Decoupled Mode - Medium, accessed August 19, 2025, https://medium.com/@ApacheDoris/slash-your-cost-by-90-with-apache-doris-compute-storage-decoupled-mode-898116c268d0
21. How the Apache Doris Compute-Storage Decoupled Mode Cuts 70% of Storage Costs—in 60 Seconds - YouTube, accessed August 19, 2025, https://www.youtube.com/watch?v=zGaVcAOp7yE
22. Optimizing Analytical Workloads: Exploring the Columnar Storage Architecture of Apache Doris - Dataprophesy, accessed August 19, 2025, https://dataprophesy.com/optimizing-analytical-workloads-exploring-the-columnar-storage-architecture-of-apache-doris/
23. Introduction to Apache Doris Storage Layer Design — Explanation of Storage Structure Design - Medium, accessed August 19, 2025, https://medium.com/@ApacheDoris/introduction-to-apache-doris-storage-layer-design-explanation-of-storage-structure-design-83f083acddce
24. Apache Doris Storage Layer Design and Storage Structure - DZone, accessed August 19, 2025, https://dzone.com/articles/analysis-of-storage-structure-design-one-of-apache
25. Hybrid Row-Columnar Storage - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/row-store
26. High-Concurrency Point Query Optimization - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/high-concurrent-point-query/
27. Table Model Overview - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/data-model/overview/
28. Best Practices - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/table-design/best-practice/
29. Integrate Apache Doris Into Your Data Architecture: Real-time Data Warehousing, accessed August 19, 2025, https://hackernoon.com/integrate-apache-doris-into-your-data-architecture-real-time-data-warehousing

30. Automatic and flexible data sharding: Auto Partition in Apache Doris - Medium, accessed August 19, 2025, https://medium.com/@ApacheDoris/automatic-and-flexible-data-sharding-auto-partition-in-apache-doris-486c1ee8b64c
31. Data Distribution Concept - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/3.0/table-design/data-partitioning/data-distribution
32. Data Distribution Concept - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/3.0/table-design/data-partitioning/data-distribution/
33. Manual partitioning - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/data-partitioning/manual-partitioning/
34. Data Bucketing - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/data-partitioning/data-bucketing/
35. Manual bucketing - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/data-partitioning/manual-bucketing/
36. What Exactly Are Tablets and Buckets in Apache Doris, What Is the Difference, and How to Configure Them Correctly? | by Gerald Li | Medium, accessed August 19, 2025, https://medium.com/@yyli810222/what-exactly-are-tablets-and-buckets-in-apache-doris-what-is-the-difference-and-how-to-configure-c9dc655f85d8
37. Deep Dive into the Apache Doris Optimizer: Unveiling the Magic with Examples, accessed August 19, 2025, https://dataprophesy.com/deep-dive-into-the-apache-doris-optimizer-unveiling-the-magic-with-examples/
38. Runtime Filter - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/query/join-optimization/runtime-filter/
39. Parallel Execution - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/optimization-technology-principle/pipeline-execution-engine/
40. A Glimpse of the Next generation Analytical Database · apache/doris Wiki - GitHub, accessed August 19, 2025, https://github.com/apache/doris/wiki/A-Glimpse-of-the-Next-generation-Analytical-Database
41. Query Analysis - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/query/query-analysis/query-analytics/
42. EXPLAIN - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/sql-manual/sql-statements/data-query/EXPLAIN/
43. Introduction to Apache Doris: A Next-Generation Real-Time Data Warehouse | HackerNoon, accessed August 19, 2025, https://hackernoon.com/introduction-to-apache-doris-a-next-generation-real-time-data-warehouse
44. Stream Load - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/data-operate/import/import-way/stream-load-manual
45. Broker Load - Apache Doris, accessed August 19, 2025,

https://doris.apache.org/docs/data-operate/import/import-way/broker-load-manual
46. Implementation Design for Spark Load - Apache Doris, accessed August 19, 2025, https://doris.apache.org/community/design/spark_load/
47. Broker Load - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/data-operate/import/broker-load-manual/
48. Routine Load - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/data-operate/import/routine-load-manual/
49. Insert Into - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/data-operate/import/insert-into-manual/
50. doris.apache.org, accessed August 19, 2025, https://doris.apache.org/community/design/spark_load/#:~:text=Doris%20supports%20various%20data%20ingestion,time%20data%20ingestion%20into%20Doris.
51. Understanding data compaction in 3 minutes - Apache Doris, accessed August 19, 2025, https://doris.apache.org/blog/Understanding-Data-Compaction-in-3-Minutes/
52. Loading Overview - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/data-operate/import/load-manual/
53. Load High Availability - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/dev/data-operate/import/load-high-availability/
54. Concurrency Control for Updates in the Primary Key Model - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/data-operate/update/unique-update-concurrent-control/
55. Concurrency Control and Queuing - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/admin-manual/workload-management/concurrency-control-and-queuing/
56. Load Best Practices - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/data-operate/import/load-best-practices/
57. High Concurrency LOAD Optimization(Group Commit) - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/data-operate/import/group-commit-manual/
58. Materialized View - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/2.0/query/view-materialized-view/materialized-view/
59. Materialized View Overview - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/materialized-view/overview/
60. Sync-Materialized View - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/dev/query-acceleration/materialized-view/sync-materialized-view
61. Overview of Asynchronous Materialized Views - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/materialized-view/async-materialized-view/overview/
62. Creating, Querying, and Maintaining Asynchronous Materialized Views - Apache

Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/materialized-view/async-materialized-view/functions-and-demands/
63. Optimizing Table Index Design - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/query-acceleration/tuning/tuning-plan/optimizing-table-index/
64. Index Overview - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/index/index-overview
65. BloomFilter Index - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/index/bloomfilter/
66. Bitmap Index - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/table-design/index/bitmap-index/
67. Metadata Cache - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/lakehouse/meta-cache/
68. Building the next-generation data lakehouse: 10X performance - Apache Doris, accessed August 19, 2025, https://doris.apache.org/blog/Building-the-Next-Generation-Data-Lakehouse-10X-Performance/
69. Apache Doris vs Trino / Presto, accessed August 19, 2025, https://doris.apache.org/docs/dev/gettingStarted/alternatives/alternative-to-trino/
70. Tuning Process - Apache Doris, accessed August 19, 2025, https://doris.apache.org/docs/3.0/query-acceleration/performance-tuning-overview/tuning-process/
71. It's 2025: How Do You Choose Between Doris and ClickHouse? | by Zen - Medium, accessed August 19, 2025, https://medium.com/@ith321.vip/its-2025-how-do-you-choose-between-doris-and-clickhouse-7d98456d9199
72. A Deep Dive into Apache Doris vs. ClickHouse - DZone, accessed August 19, 2025, https://dzone.com/articles/apache-doris-vs-clickhouse-real-time-analytics
73. Apache Druid, TiDB, ClickHouse, or Apache Doris? A Comparison of OLAP Tools | HackerNoon, accessed August 19, 2025, https://hackernoon.com/apache-druid-tidb-clickhouse-or-apache-doris-a-comparison-of-olap-tools
74. Difference between Starrocks and Doris : r/dataengineering - Reddit, accessed August 19, 2025, https://www.reddit.com/r/dataengineering/comments/134s3hl/difference_between_starrocks_and_doris/
75. Apache Doris or StarRocks? Here's what you should know - Medium, accessed August 19, 2025, https://medium.com/@ApacheDoris/apache-doris-or-starrocks-heres-what-you-should-know-f8a3573e6b04
76. FAQ: Apache Doris vs. StarRocks - Presentations, Articles and Webinars, accessed August 19, 2025, https://forum.starrocks.io/t/faq-apache-doris-vs-starrocks/128
77. Compare Apache Doris vs. Trino in 2025 - Slashdot, accessed August 19, 2025,

https://slashdot.org/software/comparison/Apache-Doris-vs-Trino/
78. Comparing Trino, ClickHouse, and Apache Doris - CloudThat, accessed August 19, 2025, https://www.cloudthat.com/resources/blog/comparing-trino-clickhouse-and-apache-doris
79. Apache Doris vs. Trino Comparison - SourceForge, accessed August 19, 2025, https://sourceforge.net/software/compare/Apache-Doris-vs-Trino/
80. Another big leap: Apache Doris 2.1.0 is released, accessed August 19, 2025, https://doris.apache.org/blog/release-note-2.1.0/