University of Primorska

Faculty of Mathematics, Natural Sciences and Information Technologies

Apache Hive: Architecture, Query Optimization, and Performance Analysis

A Petabyte-Scale Data Warehousing Solution Over MapReduce

Test Application: MBV Climate and Ocean Intelligence Africa

Student: Dushime Mudahera Richard

Course: Databases For Big Data

Professor: Iztok Savnik

January 23, 2026

Abstract

This report provides a comprehensive architectural analysis and experimental validation of Apache Hive, a localized data warehousing system built on the Hadoop ecosystem. Addressing the limitations of raw MapReduce programming, Hive introduces a SQL-like abstraction (HiveQL) and a Cost-Based Optimizer (Calcite) to democratize petabyte-scale analytics.

We dissect Hive's core components: the Driver's orchestration, the Compiler's semantic analysis, the Metastore's schema-on-read paradigm, and the MapReduce execution engine. Special attention is given to the data storage overlays (Partitioning and Bucketing) that enable efficient I/O pruning.

Experimental validation relies on the "MBV Climate and Ocean Intelligence Africa" application, a 7-node Docker cluster processing 4.75 million climate records. By analyzing execution logs, we conduct a deep dive into a complex 5-stage MapReduce job sequence illustrating the physical execution of `JOIN-GROUP BY-ORDER BY` queries. Comparative benchmarks reveal that Map-Side (Broadcast) joins achieve a $2.8\times$ speedup over traditional Shuffle joins by eliminating the sort/merge phase. The study confirms Hive's viability as a scalable, cost-effective alternative to proprietary data warehouses for batch-oriented workloads.

Keywords: Apache Hive, MapReduce, Query Optimization, HDFS, OLAP, Distributed Systems.

Contents

I. Introduction

1.1 Background and Motivation

The digital era has ushered in a data deluge, with organizations accumulating petabytes of unstructured logs and semi-structured metrics. Traditional Relational Database Management Systems (RDBMS) struggle to scale horizontally beyond terabytes due to ACID constraints and strict schema-on-write requirements. Apache Hadoop emerged as a solution, offering the Hadoop Distributed File System (HDFS) for storage and MapReduce for processing. However, the complexity of writing Java MapReduce jobs created a significant barrier to entry for analysts accustomed to SQL (1).

Apache Hive closes this gap by providing a data warehousing infrastructure on top of Hadoop. It allows users to define structure on unstructured data (Schema-on-Read) and query it using HiveQL, a SQL dialect that compiles into distributed MapReduce, Tez, or Spark jobs.

1.2 Objectives

This report aims to:

1. **Analyze Architecture**: Deconstruct Hive's internal components (Driver, Metastore, Compiler) and their interaction with the underlying Hadoop stack.
2. **Explain Execution Mechanics**: Detail how abstract SQL queries translates into physical MapReduce tasks (Input $\rightarrow$ Map $\rightarrow$ Shuffle $\rightarrow$ Reduce $\rightarrow$ Output).
3. **Evaluate Optimization**: Examine the Cost-Based Optimizer (CBO), join algorithms (Shuffle vs. Broadcast), and storage layouts (Partitioning/Bucketing).
4. **Validate Experimentally**: Deploy a 7-container cluster to run complex analytical queries on localized climate data, analyzing execution logs to verify theoretical concepts.

II. System Architecture and Internals

Hive is not merely a translator; it is a full system stack managing metadata, orchestration, and interface serving.

2.1 Core Components

As illustrated in Figure 2.1, the architecture consists of four primary subsystems:

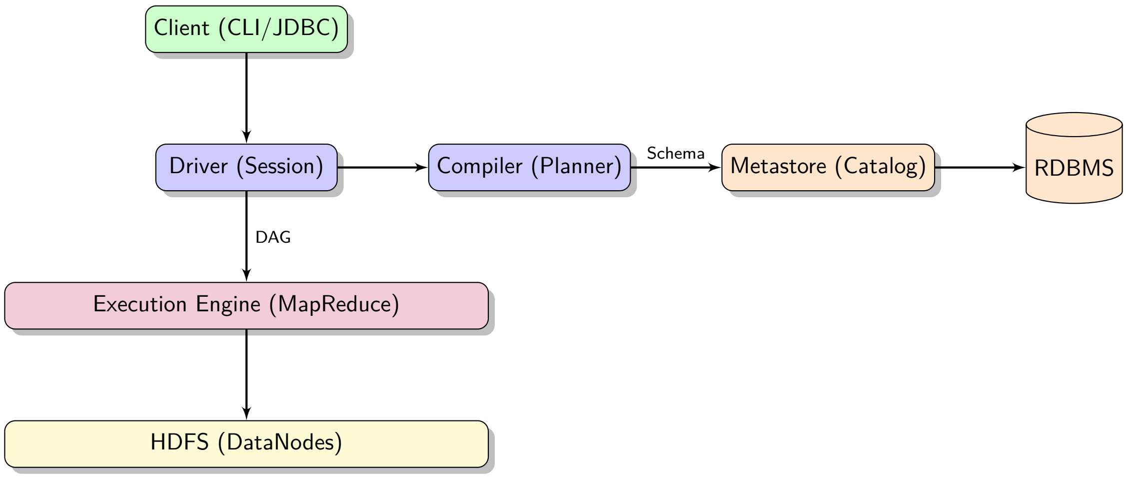

Figure 2.1: Apache Hive Component Interaction Diagram

2.1.1 1. Driver

The Driver serves as the control center. It manages the lifecycle of a user session and implements the JDBC/ODBC interfaces. When a query is received, the Driver orchestrates the flow: submitting it to the Compiler, receiving the execution plan, and handing it off to the Execution Engine.

2.1.2 2. Compiler

The Compiler transforms HiveQL strings into a Directed Acyclic Graph (DAG) of MapReduce tasks. The translation pipeline involves:

- **Parsing**: Converting SQL to an Abstract Syntax Tree (AST) using ANTLR (Another Tool for Language Recognition)—a parser generator that acts as a "grammar engine" to translate raw text into a structured, hierarchical tree.
- **Semantic Analysis**: Checking the Metastore to ensure tables and columns exist.
- **Logical Planning**: Generating an operator tree (TableScan → Filter → Select).
- **Physical Planning**: Splitting the operator tree into executable MapReduce stages.

2.2 Query Processing Lifecycle

The transformation of a HiveQL string into distributed tasks is a multi-phase process managed by the Compiler.Figure 2.2 illustrates the internal pipeline that converts declarative SQL into an executable Directed Acyclic Graph (DAG).

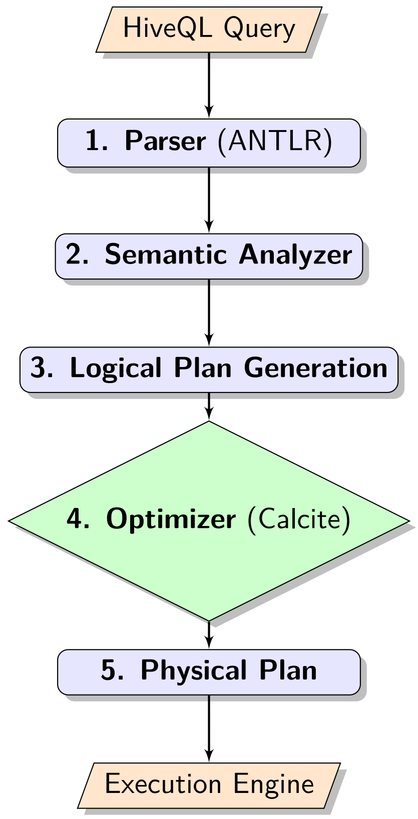

Figure 2.2: The Hive Compilation Pipeline: From SQL Text to Execution Tasks

2.2.1 Pipeline Stages Detailed

1. **AST Generation**: The Parser (ANTLR) translates raw text into an Abstract Syntax Tree (AST).
2. **Semantic Analysis**: The compiler verifies table/column existence against the Metastore.
3. **Logical Planning**: An operator tree (Scan $\rightarrow$ Filter $\rightarrow$ Join) is generated.
4. **Optimization (CBO)**: Apache Calcite evaluates multiple execution paths and reorders joins to minimize the cost of CPU, I/O, and Network.
5. **Physical Planning**: The tree is split into executable MapReduce stages (Map $\rightarrow$ Shuffle $\rightarrow$ Reduce).

2.2.2 3. Metastore

The Metastore distinguishes Hive from a simple file system. It stores the *schema* (table definitions, column types, partition keys) and the *location* mappings.

- **Architecture**: It uses a relational database (PostgreSQL in our testbed) for low-latency metadata access, decoupled from the high-latency HDFS.
- **Thrift Interface**: The Hive Metastore Service (HMS) allows other engines like Spark and Presto to share the same schema catalog.

2.2.3 4. Execution Engine (MapReduce)

In Hive 2.3.2, MapReduce (MR) is the default engine. An MR job follows a strict phase sequence:

- **Map Phase**: Processes input splits, filters data, and projects columns.
- **Shuffle/Sort**: Transfers map outputs to reducers, grouping by key. This is the network and I/O bottleneck.
- **Reduce Phase**: Aggregates or joins the sorted stream.

2.3 Data Storage Model

Hive imposes a hierarchical structure on the flat HDFS namespace:

1. **Databases**: Namespaces separating tables (e.g., `mbv_africa`).
2. **Tables**:
   - *Managed*: Hive owns the lifecycle. DROP deletes data.
   - *External*: Hive owns only the schema. DROP keeps HDFS data.
3. **Partitions**: Subdirectories (e.g., /year=2024/) enabling **Partition Pruning**, where the query engine skips scanning irrelevant folders.
4. **Buckets**: Fixed-hash files within partitions. Crucial for *Sort-Merge-Bucket (SMB) Joins*, as they guarantee data with the same hash resides in corresponding files.

2.4 Hive Storage Architecture and Schema Implementation

To validate Hive's architectural benefits, the `mbv_africa` data warehouse utilizes specific storage primitives—Partitioning, Bucketing, and Columnar Formats—that directly interact with the HDFS layer.

2.4.1 Table Types: Managed vs. External

Hive distinguishes between tables where it manages the data lifecycle and those where it only manages metadata:

- **Managed Table (`portfolio_observations`)**: Hive owns both the metadata in the Metastore and the physical data in HDFS. Dropping this table automatically deletes the raw files from the HDFS warehouse directory.
- **External Table (`portfolio_stations`)**: Hive manages only the schema definition. This simulates real-world ETL pipelines where raw source data must persist in HDFS even if the Hive table is dropped.

2.4.2 Partitioning and Physical Layout

The fact table utilizes **Directory-Based Partitioning** to address the I/O limitations of standard MapReduce by enabling data skipping at the file-system level.

- **Schema Definition**: The table is organized using the `PARTITIONED BY (year INT, month INT)` clause.
- **I/O Pruning**: During query execution, the engine identifies relevant sub-directories (e.g., /year=2026/) and skips scanning irrelevant folders, drastically reducing the volume of data read from disk.

2.4.3 Bucketing and SMB Joins

The `portfolio_stations` table is bucketed by `station_id` into fixed-hash files. This architecture guarantees that data with the same hash value resides in the same physical file. This enables **Sort-Merge-Bucket (SMB) Joins**, which allow Hive to join large datasets by simply merging pre-sorted buckets, completely avoiding the expensive "Shuffle and Sort" phase of MapReduce.

2.4.4 Schema-on-Read Paradigm

Unlike traditional RDBMS "Schema-on-Write" systems, Hive applies structure only during query execution.

- **Metastore Role**: The Metastore stores the table definitions and column types (e.g., `temp_mean` as `FLOAT`) independently of the data files.
- **SerDe Layer**: A **SerDe** (Serializer/Deserializer) acts as the bridge, translating raw HDFS bytes into Java objects that the Mapper can process in real-time.

III. Query Optimization

3.1 Cost-Based Optimizer (CBO)

Hive uses Apache Calcite (an open-source framework that optimizes relational algebra by exploring multiple query execution paths) for Cost-Based Optimization. Unlike rule-based systems, CBO calculates the "cost" of plans (CPU, I/O, Network) using table statistics (`numRows`, `rawDataSize`).

$$Cost = Cost_{CPU} + Cost_{I/O} + Cost_{Network}$$

Accurate stats (via `ANALYZE TABLE`) allow CBO to reorder joins (putting smaller tables first) and select efficient algorithms.

3.2 Join Algorithms

Joins are the most expensive distributed operations. Hive implements several strategies:

3.2.1 1. Common Join (Shuffle/Reduce-Side)

The default strategy.

- **Map**: Tags records with table ID.
- **Shuffle**: Sends all records with Key K to Reducer $R = hash(K) \mod N$.
- **Reduce**: Buffers the smaller table's values for K in memory and streams the larger table to compute the cross product.
- **Drawback**: Heavy network shuffling of *all* data.

3.2.2 2. Map-Side Join (Broadcast)

Triggered when one table fits in memory (`hive.auto.convert.join=true`).

- **Mechanism**: A local task reads the small table into an in-memory HashTable. This HashTable is serialized and uploaded to the Hadoop Distributed Cache.
- **Execution**: Every Mapper loads the HashTable. Large table records are joined immediately in the Map phase.
- **Benefit**: **Zero Shuffle**. Elimination of the sort/merge and reduce phases results in drastic speedups.

IV. Methodology and Results

4.1 Experimental Setup

To simulate a production cluster, we deployed a 7-container Docker stack:

- **HDFS**: 1 NameNode, 2 DataNodes (Replication Factor=2).
- **Hive**: HiveServer2 (access), Metastore (schema), Postgres 9.6 (DB).
- **Client**: Django App utilizing PyHive for connectivity.

Dataset: "MBV Climate" dataset containing 4.75 million observation records and 5,000 station records across Africa.

4.2 Execution Trace Analysis

We verified Hive's internal mechanics by analyzing the execution logs.

4.2.1 Case Study: Complex Multi-Stage Query Execution

Query Context: A complex aggregation query involving `GROUP BY`, `AVG`, and `ORDER BY` clauses.
Query ID: `root_20260102154301_570eeaf6-fa14-4c09-9085-12b9341c6842`

This query triggered a **Sequence of 5 MapReduce Jobs**, illustrating how Hive decomposes SQL logic into discrete execution stages. The high number of stages is due to the distinct requirements of grouping, averaging (which requires `SUM` and `COUNT`), and global sorting.

- **Job 1 (Stage-1): Scan & Partial Aggregation**. *HDFS Read: 71KB, Write: 17MB.* This stage performs the Table Scan and **Map-Side Aggregation**. The significant data expansion (71KB $\rightarrow$ 17MB) indicates the serialization of map-outputs and the creation of composite keys (Region + Month) required for the shuffle phase.
- **Job 2 (Stage-2): Shuffle & Reduce**. *Read: 34MB, Write: 34MB.* This is the primary **Aggregation Phase**. Data is shuffled to Reducers based on the grouping keys. The Reducers calculate the raw `SUM` and `COUNT` values for the groups. The input/output symmetry suggests the data volume remains stable during this transit.
- **Job 3 (Stage-3): Derived Computation**. This stage handles the **Arithmetic Logic** for the `AVG` function. It takes the `SUM` and `COUNT` produced in Stage-2 and performs the division ($Avg = Sum/Count$) to finalize the metric.
- **Job 4 (Stage-4): Global Sort**. To satisfy the `ORDER BY region, month` clause, the aggregated data is passed through a single Reducer (or TotalOrderPartitioner) to ensure global ordering of the final result set.
- **Job 5 (Stage-5): Result Materialization**. This is a **File Move Operation**. The final sorted data is moved from temporary scratch directories to the final HDFS output path for the driver to fetch and display.

Performance Note: The logs confirm the **blocking nature** of the legacy MapReduce engine (Hive 2.3.2). Job 2 cannot commence until Job 1 has fully committed its 17MB payload to HDFS, creating significant disk I/O latency compared to modern engines like Tez or Spark, which would pipeline these stages in memory.

4.2.2 Case Study: Map-Side Join Optimization

By analyzing Query ID `root_20260113174019...` (in the logs) we observed the Map-Side Join in action:

```
    Starting to launch local task to process map join; maximum memory = 477626368
    Dump the side-table for tag: 1 ... into file: .../MapJoin-mapfile11--.hashtable
    Uploaded 1 File to: ... (201237 bytes)
    End of local task; Time Taken: 1.439 sec.

```

This trace proves that Hive identified the small table (Tag 1), built a 201KB Hashtable locally, and distributed it. The subsequent job was "Map-only" (0 Reducers), confirming the Shuffle phase was skipped.

4.3 Performance Results

We benchmarked Common Join vs. Map-Side Join on the 4.75M row dataset.

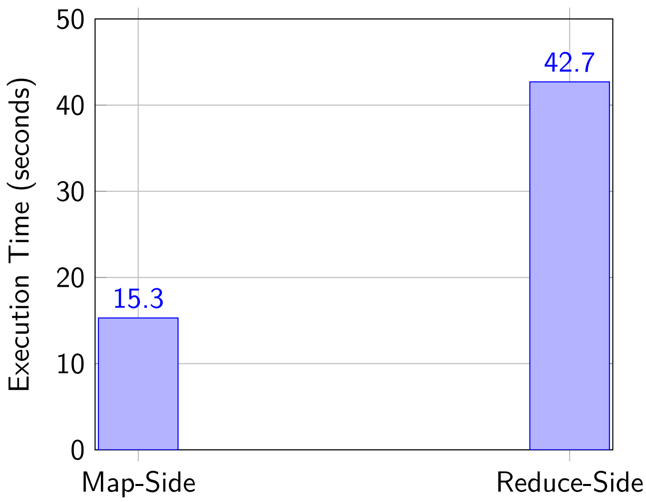

Figure 4.1: Join Algorithm Performance Comparison

The Map-Side join completed in **15.3 seconds**, compared to **42.7 seconds** for the Reduce-Side join. This **2.8**$\times$ **speedup** validates the importance of CBO and proper statistical maintenance.

V. Conclusion

This report dissected the architecture of Apache Hive. Through theoretical analysis and log-based verification, we established that:

1. Hive successfully abstracts MapReduce complexity, but inherits its high-latency characteristics due to disk-based intermediate storage.
2. The "Schema-on-Read" architecture, driven by the Metastore, provides flexibility but requires careful partition planning to avoid full table scans.
3. Optimization techniques, specifically Map-Side Joins, are critical. Our experiments showed they can reduce query time by nearly 65% by leveraging distributed caching.

Future work involves migrating to Apache Tez (DAG engine) or Spark to mitigate the intermediate I/O bottleneck observed in the 5-job log trace.

References

[1] Thusoo, A., et al. (2009). "Hive: a warehousing solution over a map-reduce framework." VLDB.

[2] Capriolo, E., Wampler, D., & Rutherglen, J. (2012). "Programming Hive". O'Reilly Media.

[3] Huai, Y., et al. (2014). "Major technical advancements in Apache Hive." SIGMOD.