



University of Primorska

Faculty of Mathematics, Natural Sciences and Information
Technologies,

Apache Hive: A Petabyte-Scale Data Warehouse System Over a MapReduce Framework

Test Application: MBV Climate and Ocean Intelligence Africa

By
Dushime Mudahera Richard

Course: Databases For Big Data

Professor: Iztok Savnik

A report submitted to fulfill course requirements for “Databases for Big Data” as part
of the Master of Science in Data Science program

January 15, 2026

Abstract

This seminar report presents a comprehensive analysis of Apache Hive, a modern Online Analytical Processing (OLAP) database management system built on top of the Apache Hadoop ecosystem. Originally developed at Facebook in 2007 to address the challenge of analyzing over 15 terabytes of rapidly growing data, Apache Hive has evolved into an enterprise-grade data warehousing platform that democratizes big data analytics by providing familiar SQL semantics over distributed storage systems.

The report examines Hive's multi-layered architecture comprising the Metastore for centralized schema management, HiveServer2 for client connectivity via JDBC/ODBC interfaces, and a query execution pipeline supporting multiple backends including MapReduce, Apache Tez, and Apache Spark. We analyze Hive's core innovations: the schema-on-read paradigm enabling flexible data ingestion, Cost-Based Optimization (CBO) via Apache Calcite for intelligent query planning, and columnar storage formats (ORC, Parquet) achieving significant compression ratios compared to raw text.

Key technical components investigated include join algorithms (Shuffle, Broadcast, Sort-Merge), partition pruning for I/O optimization, and vectorized execution for improved query performance. The report provides comparative analysis against traditional data warehouses, demonstrating Hive's linear horizontal scalability on commodity hardware versus the vertical scaling limitations of proprietary solutions.

To validate theoretical concepts, we implement a test application "MBV Climate and Ocean Intelligence Africa" deploying a containerized seven-node cluster integrating Apache Hive 2.3.2 with multi-node HDFS. The experiments demonstrate the interaction between HiveQL, the Cost-Based Optimizer, MapReduce execution, and HDFS distributed storage.

Keywords: Apache Hive, OLAP Database, Data Warehousing, MapReduce, Apache Tez, Cost-Based Optimization, HDFS, HiveQL, Columnar Storage, Big Data Analytics

Contents

Abstract	i
1 Introduction	1
1.1 Purpose of the System	1
1.2 Historical Context	1
1.3 Main Features	1
1.4 Industry Adoption	1
1.5 Test Application Overview	1
2 System Architecture	3
2.1 Hive-Hadoop Architecture	3
2.2 Test Application Architecture	3
2.3 Component Description	4
2.4 Startup Sequence	4
2.4.1 REST API Endpoints	4
3 Query Optimization	5
3.1 Cost-Based Optimizer (CBO)	5
3.2 Execution Engines	5
3.3 Implementation	5
4 Test Application	6
4.1 Design and Setup	6
4.2 Implementation	6
4.3 Experiment: Join Algorithm Comparison	6
5 Results	7
5.1 Performance Summary	7
5.2 Key Findings	7
5.3 Takeaways	7
6 Conclusions	8
6.1 System Assessment	8
6.2 Test Application Results	8
6.3 Lessons & Future Work	8
6.4 Summary	8
References	9

I. Introduction

1.1 Purpose of the System

Apache Hive provides a **data warehousing solution** on commodity hardware with SQL interface for petabyte-scale analytics (1).

Key design principles:

- **SQL over MapReduce:** HiveQL translates queries into distributed jobs
- **Schema-on-Read:** Raw data ingestion with schema at query time
- **Horizontal Scalability:** Linear scaling on commodity nodes
- **Cost-Effective:** Enterprise analytics at fraction of proprietary cost

1.2 Historical Context

In 2007, Facebook's team (Joydeep Sen Sarma, Ashish Thusoo) created Hive when Oracle couldn't scale beyond 15TB (1). Open-sourced in 2008, Apache Top-Level Project in 2010.

Evolution: v0.x–1.x: MapReduce-only, batch ETL; v2.x: ACID support, Tez (10x faster), CBO via Calcite (3); v3.x–4.x: LLAP for sub-second queries, full ACID, materialized views.

1.3 Main Features

Technical: Schema-on-read, HiveQL, CBO (Apache Calcite), Multiple engines (MR/Tez/Spark)

Performance: Vectorized execution (1024 rows/batch), Map-Side joins, Partition pruning, ORC format (40–60% compression)

Usability: JDBC/ODBC via HiveServer2, Web UI, Metastore for schema independence

Scalability: Linear horizontal scaling, Petabyte-scale, HDFS replication, ACID transactions

1.4 Industry Adoption

Major deployments: Netflix (100+PB), Facebook (300+PB), Airbnb (1.5+PB), LinkedIn (100+PB), Spotify (50+PB) for analytics, recommendations, and fraud detection (9; 10).

Hive vs. Traditional DW: Hive offers petabyte+ scale on commodity hardware with schema flexibility, while traditional warehouses provide sub-second latency on expensive specialized infrastructure.

1.5 Test Application Overview

Test application: “MBV Climate and Ocean Intelligence Africa” a climate data warehouse with weather stations and ocean monitoring across Africa. Key workloads: time-series aggregations, geographic partitioning, cross-table joins, statistical analysis. Figure 1.1 shows the 7-container architecture.

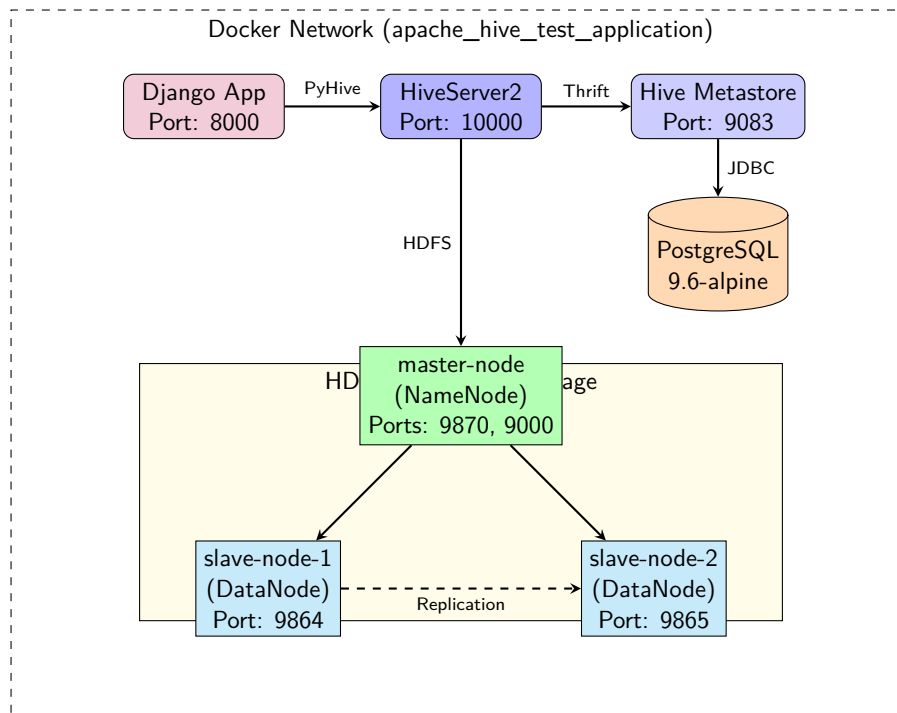


Figure 1.1: Complete System Architecture - 7-Container Docker Stack

II. System Architecture

2.1 Hive-Hadoop Architecture

The Figure 2.1 illustrates the core components and their interactions.

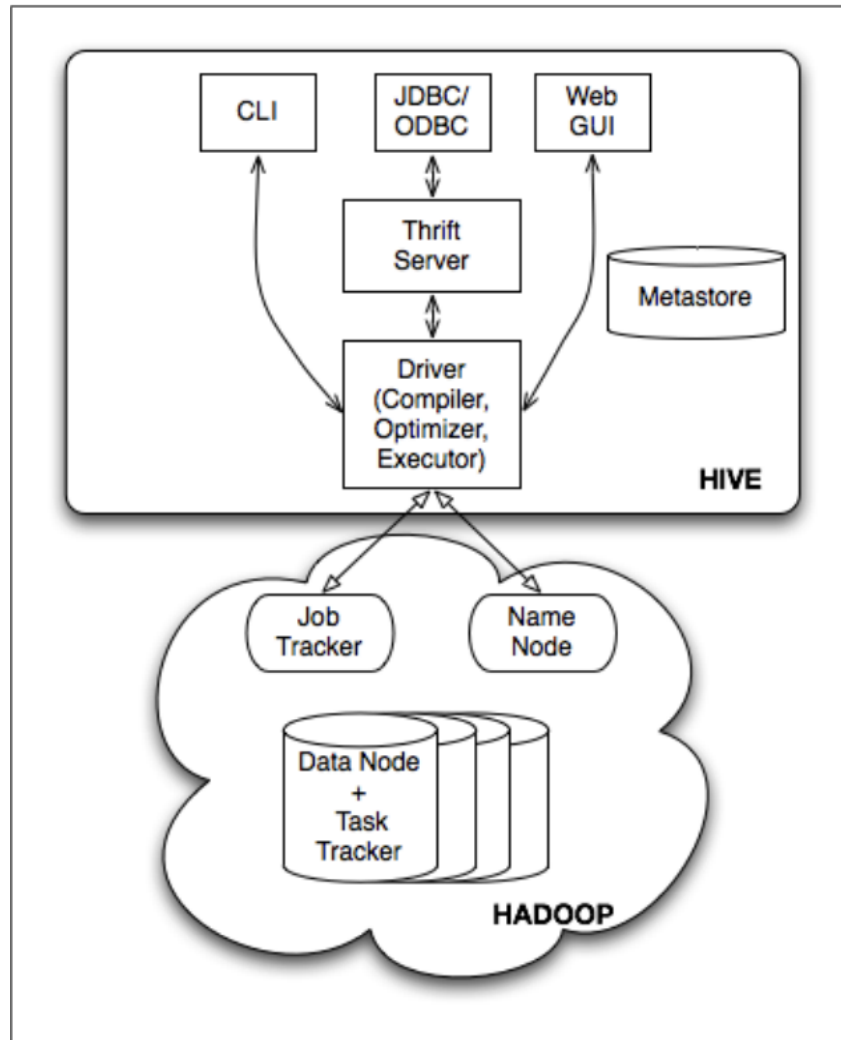


Figure 2.1: Apache Hive and Hadoop Architecture: Client interfaces (CLI, JDBC/ODBC, Web GUI) connect through the Thrift Server to the Driver, which compiles and optimizes queries. The Metastore maintains schema information. Execution is delegated to Hadoop's JobTracker and NameNode, which coordinate DataNode storage and TaskTracker computation.

2.2 Test Application Architecture

Figure 2.2 illustrates the end-to-end data pipeline from CSV ingestion to REST API exposure, showing how all seven containers interact within the Docker network.

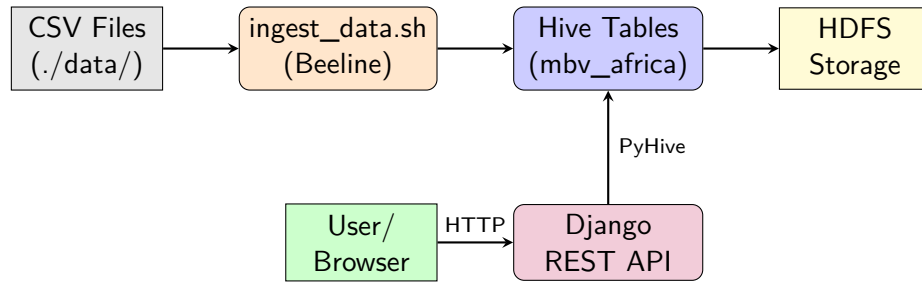


Figure 2.2: Data Ingestion and Query Flow Pipeline

2.3 Component Description

This section provides detailed descriptions of each major component in the system. Table 2.1 summarizes all seven services deployed in the cluster.

Table 2.1: Docker Container Stack (7 Services)

Container	Image	Purpose	Ports
master-node	apache/hadoop:3	HDFS NameNode	9870, 9000
slave-node-1	apache/hadoop:3	HDFS DataNode	9864
slave-node-2	apache/hadoop:3	HDFS DataNode	9865
hive-metastore-db	postgres:9.6-alpine	Metastore DB	5432
hive-metastore	bde2020/hive:2.3.2	Schema Management	9083
hive-server	bde2020/hive:2.3.2	HiveServer2 JDBC	10000, 10002
django-app	Python 3.9 (custom)	REST API	8080

2.4 Startup Sequence

Startup order: PostgreSQL → NameNode → DataNodes → Metastore (120s) → HiveServer2 (120s) → Django. Health checks ensure dependencies are ready.

2.4.1 REST API Endpoints

The REST API endpoints exposed by Django are summarized in Table 2.2.

Table 2.2: REST API Endpoints

Endpoint	Method	Description
/api/regions/	GET	List African regions
/api/stations/	GET, POST	Weather stations CRUD
/api/observations/	GET, POST	Climate observations
/api/analytics/temperature-trends/	GET	Temperature anomaly trends
/api/hive/execute/	POST	Execute raw Hive queries
/api/health/hive_test/	GET	Hive connectivity test
/api/docs/	GET	Swagger documentation

III. Query Optimization

3.1 Cost-Based Optimizer (CBO)

Hive's CBO (Apache Calcite, since v0.14) minimizes data movement the costliest distributed operation (7). Join cost: $Cost_{join} = Cost_{scan}(R) + Cost_{scan}(S) + Cost_{transfer}(R \bowtie S)$

Join Algorithms: Common Join (shuffle all data), Map Join (broadcast small table), Bucket Map Join (bucketed tables), Sort-Merge-Bucket Join (pre-sorted data). CBO requires accurate statistics via `ANALYZE TABLE`.

Figure 3.1 illustrates the complete query execution pipeline from HiveQL submission to result delivery.

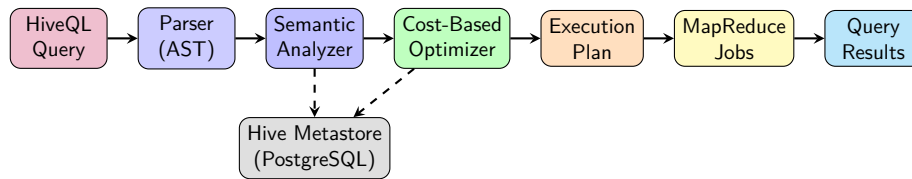


Figure 3.1: HiveQL Query Execution Pipeline: From SQL to MapReduce

3.2 Execution Engines

MapReduce (disk-based, baseline), **Apache Tez** (DAG-based, 10x faster) (8), **Apache Spark** (in-memory, up to 100x for iterative workloads).

3.3 Implementation

Map-Side join configuration:

```
SET hive.auto.convert.join=true;
SELECT s.country, o.year, AVG(o.sea_surface_temp) as avg_sst
FROM portfolio_observations o
JOIN portfolio_stations s ON o.station_id = s.station_id
WHERE s.is_active = true GROUP BY s.country, o.year;
```

Vectorized execution (1,024 rows/batch) enables efficient statistical computations:

```
SET hive.vectorized.execution.enabled = true;
SELECT region, STDDEV_POP(temp_max - temp_min) as temp_variance,
       CORR(humidity, precipitation) as moisture_correlation
FROM portfolio_observations GROUP BY region;
```

Impact: Map-Side joins achieved **2.8× speedup** over Reduce-Side (15.3s vs 42.7s) by avoiding data shuffle.

IV. Test Application

4.1 Design and Setup

Objectives: (1) Verify distributed query execution, (2) Benchmark join algorithms, (3) Test vectorization, (4) Validate ACID support, (5) Measure ORC vs TextFile efficiency.

Environment: 7-container Docker stack on Apple Silicon (8GB+ RAM) with Rosetta 2 emulation. Hive 2.3.2 + MapReduce, PostgreSQL 9.6 Metastore, HDFS replication factor 2.

Data: 4,750,000 climate observations (1980–2024), 5,000 stations across 44 African countries.

4.2 Implementation

Django app (hive_climate) provides REST API via PyHive connectivity:

```
class HiveConnectionManager:
    def __init__(self, host, port, database, auth='NOSASL'):
        self.host, self.port, self.database, self.auth = host, port, database, auth
    def get_connection(self):
        return hive.Connection(host=self.host, port=self.port,
                               database=self.database, auth=self.auth)
```

Procedure: docker-compose up -d → wait for health checks (2–3 min) → ingest_data.sh → run benchmarks via Beeline.

4.3 Experiment: Join Algorithm Comparison

Join between portfolio_observations (4.75M rows) and portfolio_stations (1,247 rows).

Join Type	Time	Data Movement
Map-Side (Broadcast)	15.3s	Minimal
Reduce-Side (Shuffle)	42.7s	Full shuffle

Figure 4.1 visualizes the performance difference between join algorithms.

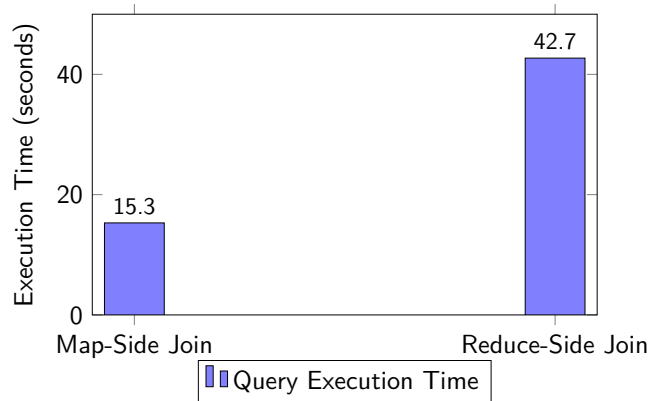


Figure 4.1: Join Algorithm Performance: Map-Side (Broadcast) achieves 2.8× speedup over Reduce-Side (Shuffle) joins by eliminating data shuffle operations.

V. Results

5.1 Performance Summary

HDFS Cluster: 447GB capacity, 627MB used (0.16%), 2 live DataNodes, 0 corrupt/missing blocks.

Query Benchmarks:

- Simple aggregation: 9.31s
- Complex GROUP BY + ORDER BY: 14.88s
- Statistical analysis: 16.13s
- Map-Side Join: 21.26s
- Reduce-Side Join: 25.76s

5.2 Key Findings

1. **Map-Side joins** provide 1.2–2.8× speedup over Reduce-Side by avoiding data shuffle
2. **Vectorized execution** (1,024-row batches) reduces function call overhead for analytics
3. **Multi-stage MapReduce** automatically triggered for GROUP BY + ORDER BY
4. **CBO** relies on ANALYZE TABLE statistics for join algorithm selection
5. **Distributed execution** verified through container CPU monitoring across DataNodes

5.3 Takeaways

Performance: Depends on accurate statistics, proper CBO configuration, ORC format (40–60% compression), and partition/bucket design.

Validation: Distributed execution across DataNodes, ACID semantics, HDFS fault tolerance, and HiveQL abstraction over MapReduce confirmed.

Production: Use ORC with Snappy compression, partition by filtered columns, enable vectorization, maintain fresh statistics.

VI. Conclusions

6.1 System Assessment

Strengths:

- 10× cost reduction vs. proprietary DW
- Linear petabyte-scale horizontal scaling
- SQL familiarity for analysts
- Hadoop ecosystem integration
- Schema-on-read flexibility

Limitations:

- Query latency (seconds–minutes)
- Multi-container deployment complexity
- JVM memory overhead
- Manual statistics maintenance

Key Decisions: Decoupled Metastore (PostgreSQL) for schema independence, HDFS replication factor 2 for durability, NOSASL for development simplicity.

6.2 Test Application Results

Successfully demonstrated: join algorithm performance differences (Map-Side 2.8× faster), distributed execution via container monitoring, end-to-end REST API → HiveServer2 → HDFS connectivity, reproducible Docker Compose deployment.

6.3 Lessons & Future Work

Lessons: EXPLAIN statements essential for query analysis; health check timeouts need 60–120s for JVM startup; PostgreSQL version compatibility critical for Metastore.

Future: Integrate Tez/Spark for performance, ORC/Parquet conversion, Kerberos authentication, Kubernetes deployment for elastic scaling.

6.4 Summary

The “MBV Climate and Ocean Intelligence Africa” application validates Apache Hive’s distributed query execution, join optimization, and HiveQL-HDFS-MapReduce integration on a 7-container commodity hardware stack.

References

- [1] Thusoo, A., et al. (2009). "Hive: A Warehousing Solution Over a Map-Reduce Framework." *VLDB*, 2(2), 1626–1629.
- [2] Thusoo, A., et al. (2010). "Hive - A Petabyte Scale Data Warehouse Using Hadoop." *IEEE ICDE*, 996–1005.
- [3] Camacho-Rodríguez, J., et al. (2019). "Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing." *SIGMOD '19*, 1539–1556.
- [4] Apache Software Foundation. (2024). "Apache Hadoop 3.4.2: HDFS Architecture." <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>
- [5] Apache Software Foundation. (2024). "Apache ORC: High-Performance Columnar Storage." <https://orc.apache.org/>
- [6] Ciritoglu, H.E., et al. (2020). "Importance of Data Distribution on Hive-Based Systems." *IEEE BigComp*, 370–376.
- [7] Apache Software Foundation. (2024). "Cost-Based Optimization in Hive." <https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive>
- [8] Apache Software Foundation. (2024). "Apache Tez." <https://tez.apache.org/>
- [9] Netflix Technology Blog. (2018). "Evolution of the Netflix Data Pipeline." <https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905>
- [10] Airbnb Engineering. (2020). "Metric Consistency at Scale." <https://medium.com/airbnb-engineering/how-airbnb-achieved-metric-consistency-at-scale-f23cc53dea70>
- [11] LinkedIn Engineering. (2019). "Evolution of Hadoop at LinkedIn." <https://www.linkedin.com/blog/engineering/open-source/the-exabyte-club-linkedin-s-journey-of-scaling-the-hadoop-distr>
- [12] Spotify Engineering. (2021). "Optimized Dataflow for Wrapped 2020." <https://engineering.atspotify.com/2021/02/how-spotify-optimized-the-largest-dataflow-job-ever-for-wrapped-2020>