SE446: Big Data Engineering

Week 4A: Introduction to Apache Hive

Prof. Anis Koubaa

SE 446
Alfaisal University
https://github.com/aniskoubaa/big_data_course

Spring 2026

جامعة الفيصل
Alfaisal University

Today's Agenda

Recap: MapReduce Was Powerful But Painful

MapReduce gave us:

- ✔ Distributed processing
- ✔ Fault tolerance
- ✔ Scalability

But writing MapReduce is:

- ✘ Verbose
- ✘ Low-level
- ✘ Slow to develop

The Pain: Too Much Code for Simple Queries

Productivity Problem

A simple "count crimes by type" query:

- MapReduce: **30+ lines** of Python
- SQL: **3 lines**

The Big Idea

What if we could write **SQL** and have it translated to MapReduce jobs?

The Motivation: SQL vs. MapReduce

MapReduce (Python):

```python
def mapper(_, line):
    fields = line.split(",")
    crime_type = fields[5]
    emit(crime_type, 1)

def reducer(crime_type, counts):
    emit(crime_type, sum(counts))
```

Plus: framework setup, job configuration, I/O handling...

HiveQL:

```sql
SELECT primary_type,
       COUNT(*) AS cnt
FROM crimes
GROUP BY primary_type
ORDER BY cnt DESC;
```

✔ That's it. Hive compiles this into MapReduce for you.

Key Insight

Hive lets **data analysts who know SQL** work with Big Data without learning MapReduce programming.

Apache Hive: Overview

Definition

Apache Hive is a **data warehouse system** built on top of Hadoop that provides:

- A **SQL-like query language** (HiveQL / HQL)
- **Schema-on-read** for data stored in HDFS
- Automatic translation of queries into **MapReduce / Tez / Spark** jobs

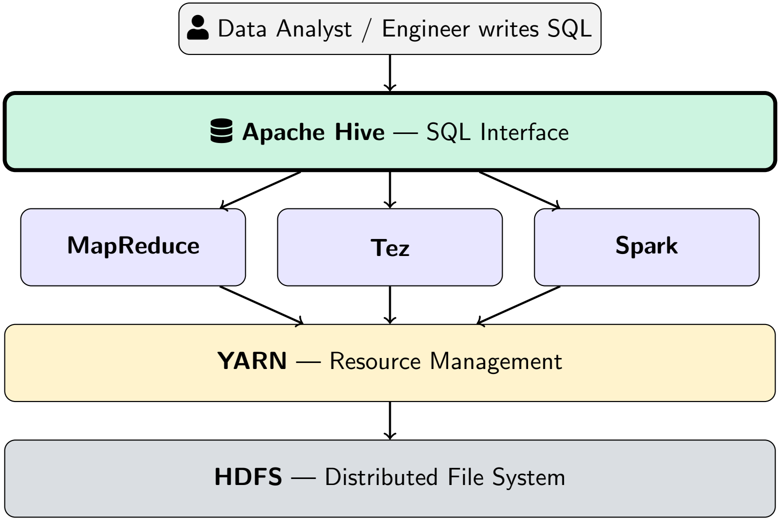
Origin:

- Developed at **Facebook** (2007)
- Open-sourced as Apache project
- Used by Facebook, Netflix, Airbnb, ...

Key idea:

- Files in HDFS $\rightarrow$ treated as tables
- SQL queries $\rightarrow$ compiled to distributed jobs
- No data movement required

Where Hive Fits in the Hadoop Ecosystem



Schema-on-Read vs. Schema-on-Write

Traditional RDBMS (Schema-on-Write)

- Define schema **before** loading data
- Data validated at **write time**
- Slow load, fast read
- Example: MySQL, PostgreSQL

Schema → Load → Query

Hive (Schema-on-Read)

- Store raw files first in HDFS
- Define schema **when reading**
- Fast load, flexible schema
- Data stays in HDFS as-is

Load → Schema → Query

Why This Matters

Schema-on-read allows you to load data **first**, ask questions **later**. Perfect for exploratory Big Data analysis where schema may evolve.

Hive Is NOT a Database

Common Misconception

Students often think Hive **is** a database. It is **not**!

What Hive IS:

- ✔ A SQL interface to HDFS
- ✔ A query compiler (SQL $\rightarrow$ MR/Tez/Spark)
- ✔ A metadata catalog (Metastore)
- ✔ Built for **batch analytics**

What Hive is NOT:

- ✘ Not for real-time queries
- ✘ Not for row-level updates
- ✘ Not a replacement for MySQL
- ✘ Not designed for < 1 sec latency

Rule of Thumb

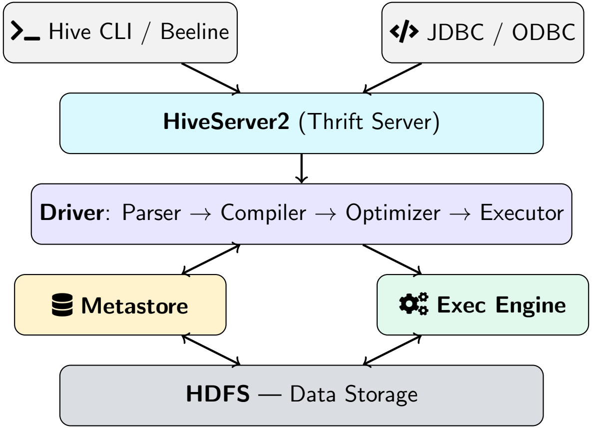
Use Hive for **analytical queries** over **large datasets** (GB to PB).
For real-time lookups, use HBase, Cassandra, or a traditional RDBMS.

Common Limitations & Gotchas

Students often encounter:

1. **Batch processing mindset**
   - Queries run as MapReduce/Tez jobs → seconds to minutes
   - Not suitable for interactive dashboards or millisecond responses
2. **CSV/delimiter issues**
   - Quoted fields, embedded commas, escape characters
   - Header rows need `skip.header.line.count`
   - NULL representation varies (empty string vs. `\N`)
3. **Partitioned table setup**
   - Must explicitly `ADD PARTITION` or use dynamic partitioning
   - Partition columns not stored in data files
4. **Schema evolution**
   - Adding columns is easy; changing types is hard
   - Use `ALTER TABLE` carefully

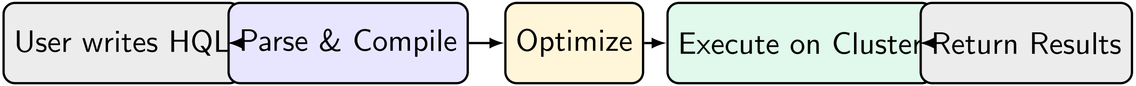
Hive Architecture: The Big Picture



Component Deep Dive

- **HiveServer2** Accepts connections from clients (Beeline, JDBC, Python). Supports concurrent users and authentication.
- **Driver** The brain of Hive:
  1. **Parser**: Converts HQL to Abstract Syntax Tree (AST)
  2. **Compiler**: Generates logical plan from AST
  3. **Optimizer**: Applies optimizations (predicate pushdown, partition pruning)
  4. **Executor**: Converts plan to physical jobs (MapReduce / Tez / Spark)
- **Metastore** Stores metadata (table names, columns, data types, partition info, file locations). Uses a relational DB (MySQL, PostgreSQL, Derby).
- **HDFS** Where the actual data lives. Hive **never moves data** — it reads in place.

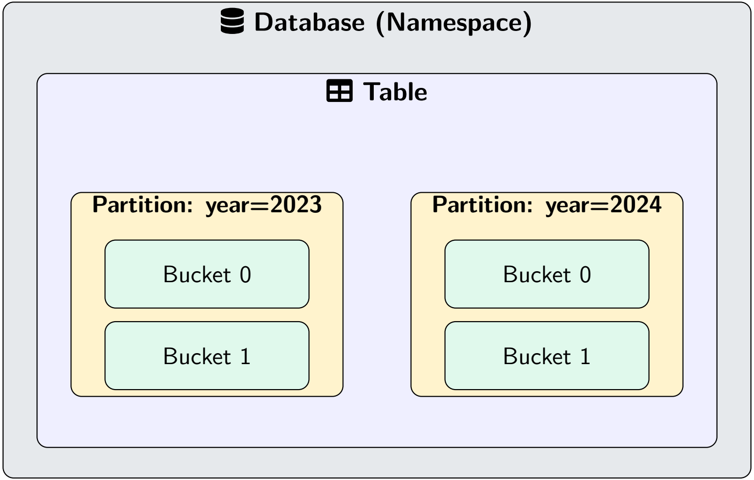
How a Hive Query Executes



Example: `SELECT COUNT(*) FROM crimes WHERE year = 2023;`

1. **Parse**: Build AST from SQL string
2. **Compile**: Determine table location, columns needed
3. **Optimize**: Prune partitions, push down predicates
4. **Execute**: Launch MapReduce/Tez job on YARN
5. **Return**: Collect results and display

Hive Data Model: Key Concepts



Data Model Hierarchy

1. **Database** — Namespace for organizing tables
   - Default database: `default`
   - Example: `CREATE DATABASE crime_analytics;`
2. **Table** — Schema definition mapped to HDFS directory
   - **Managed (Internal)**: Hive owns the data; dropping table deletes files
   - **External**: Hive only manages metadata; data remains in HDFS
3. **Partition** — Subdirectories that split data by column value
   - Physical directory: `/warehouse/crimes/year=2023/`
   - Enables **partition pruning** for faster queries
4. **Bucket** — Hash-based split within a partition
   - Improves join performance and sampling
   - Fixed number of files per partition

Managed vs. External Tables

Managed (Internal) Table

- Hive **owns** the data
- Data stored in Hive warehouse directory
- `DROP TABLE` → deletes data!
- Use for: temporary/derived tables

⚠ Careful with DROP

External Table

- Hive only manages **metadata**
- Data stays at its HDFS location
- `DROP TABLE` → data survives!
- Use for: raw data, shared datasets

Safe to drop

Best Practice

Always use **EXTERNAL** tables for raw/source data.
Use managed tables only for intermediate results that Hive can safely delete.

Where Metadata vs. Data Lives

Metastore (RDBMS)

- Table schema (columns, types)
- Partition definitions
- File locations
- SerDe configuration
- Statistics

Storage: MySQL, PostgreSQL, Derby

HDFS

- Actual data files
- CSV, ORC, Parquet, JSON
- Partition directories
- Managed tables default: `/user/hive/warehouse/`
- External tables: custom location

Key Insight

`DROP TABLE` removes **metadata** always, but only removes **data** for managed tables.

Primitive Data Types

Category	Type	Description
Numeric	`INT`	32-bit integer
Numeric	`BIGINT`	64-bit integer
Numeric	`FLOAT`	Single precision
Numeric	`DOUBLE`	Double precision
Numeric	`DECIMAL(p,s)`	Exact numeric (e.g., money)
String	`STRING`	Variable-length text
String	`VARCHAR(n)`	Variable, max length *n*
String	`CHAR(n)`	Fixed-length
Date/Time	`TIMESTAMP`	YYYY-MM-DD HH:MM:SS
Date/Time	`DATE`	YYYY-MM-DD
Boolean	`BOOLEAN`	TRUE / FALSE
Binary	`BINARY`	Byte array

Complex Data Types: ARRAY

`ARRAY<type>` — Ordered list of elements

```
-- Column definition
tags ARRAY<STRING>

-- Sample data: ["hadoop","hive","spark"]

-- Query: Access first element (0-indexed)
SELECT tags[0] FROM articles;
-- Result: "hadoop"
```

Complex Data Types: MAP

`MAP<key,value>` — Key-value pairs

```
-- Column definition
scores MAP<STRING,INT>

-- Sample data: {"midterm":85, "final":92}

-- Query: Access value by key
SELECT scores["final"] FROM students;
-- Result: 92
```

Complex Data Types: STRUCT

`STRUCT<fields>` — Named fields (like a record)

```
-- Column definition
address STRUCT<city:STRING, zip:STRING>

-- Sample data: {"Riyadh","11533"}

-- Query
SELECT address.city, address.zip
FROM employees;
-- Result: "Riyadh", "11533"
```

Why Complex Types?

Big Data often comes as JSON, logs, or nested records. Complex types let Hive handle them **natively** without flattening.

Storage Formats in Hive

Format	Type	Splittable	Compressed	Best For
TextFile	Row	✔	✖	Simple CSV/TSV
SequenceFile	Row	✔	✔	Intermediate data
ORC	Columnar	✔	✔	**Hive-optimized**
Parquet	Columnar	✔	✔	**Spark/cross-platform**
Avro	Row	✔	✔	Schema evolution

Row-based formats:

- Read entire rows
- Good for `SELECT *`
- Bad for "give me column A only"

Columnar formats (ORC/Parquet):

- Read only needed columns
- Excellent compression
- **10–100x** faster for analytics

ORC vs. Parquet: Which to Choose?

ORC (Optimized Row Columnar)

- ✔ Native to Hive
- ✔ Best compression (ZLIB/Snappy)
- ✔ Built-in indexes
- ✔ ACID transaction support
- ✘ Less portable outside Hadoop

Parquet

- ✔ Cross-platform (Spark, Presto, etc.)
- ✔ Good compression
- ✔ Nested data support
- ✔ Industry standard
- ✘ Slightly less optimized for Hive

Recommendation for This Course

Use **ORC** for Hive-only workloads, **Parquet** when data is shared with Spark.

What Is a SerDe?

SerDe = Serializer / Deserializer

Tells Hive **how to read** (deserialize) and **how to write** (serialize) data from/to files.

Common SerDes:

- `LazySimpleSerDe` — Default for CSV/TSV
- `OpenCSVSerDe` — Handles quoted CSV fields
- `JsonSerDe` — Parse JSON records
- `OrcSerDe` / `ParquetSerDe` — Columnar formats
- `RegexSerDe` — Parse via regex (log files)

SerDe Example: Parsing JSON

Scenario: You have JSON log files in HDFS.

```sql
-- Example: JSON SerDe
CREATE EXTERNAL TABLE events (
    user_id STRING, action STRING, ts TIMESTAMP
)
ROW FORMAT SERDE 'org.apache.hive.serde2.JsonSerDe'
LOCATION '/data/events/';
```

How It Works

Hive reads each line as a JSON object, extracts fields, and maps them to table columns.

Partitioning: Divide and Conquer

Idea

Split a table into **subdirectories** based on a column value. Hive reads **only** the relevant directory when filtering.

Without Partitioning:

- `/warehouse/crimes/data.csv`
- Query: `WHERE year=2023`
- ✗ Full table scan

With Partitioning:

- `/warehouse/crimes/year=2022/`
- `/warehouse/crimes/year=2023/`
- `/warehouse/crimes/year=2024/`
- ✓ Reads only year=2023/

Performance

Speeds up queries by **orders of magnitude**. Use **low cardinality** columns (year, month, country).

Creating a Partitioned Table

```sql
-- Create a partitioned external table
CREATE EXTERNAL TABLE crimes (
    case_number   STRING,
    primary_type  STRING,
    description   STRING,
    district      INT,
    arrest        BOOLEAN,
    latitude      DOUBLE,
    longitude     DOUBLE
)
PARTITIONED BY (year INT, month INT)
STORED AS ORC
LOCATION '/data/crimes/';
```

Loading Data into Partitioned Tables

```sql
-- Add partition manually (static)
ALTER TABLE crimes
ADD PARTITION (year=2023, month=6)
LOCATION '/data/crimes/year=2023/month=6';
```

Dynamic Partitioning

```sql
-- Enable dynamic partitioning
SET hive.exec.dynamic.partition.mode=nonstrict;

-- Insert with dynamic partitions
-- (Assuming crimes_staging has all columns)
INSERT OVERWRITE TABLE crimes PARTITION (year, month)
SELECT case_number, primary_type, description,
       district, arrest, latitude, longitude,
       YEAR(to_date(date_str)) AS year,
       MONTH(to_date(date_str)) AS month
FROM crimes_staging;
```

Important

Partition columns (`year`, `month`) must be **last** in SELECT.
Don't use `SELECT *` if staging table already has those columns.

Bucketing: Fine-Grained Splits

What Is Bucketing?

Divide data within a partition into a **fixed number of files** using a **hash function** on a column.

Benefits:

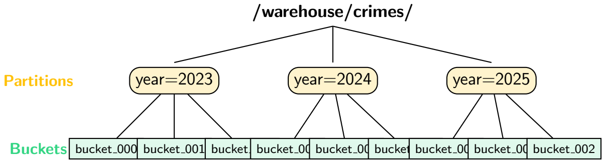
- **Efficient joins**: Matching buckets joined directly
- **Sampling**: Read 1 of N buckets for quick exploration
- **Consistent file sizes**: Better parallelism

Bucketing Example

```
CREATE TABLE crimes_bucketed (
    case_number  STRING,
    primary_type STRING,
    district     INT,
    arrest       BOOLEAN
)
CLUSTERED BY (district) INTO 8 BUCKETS
STORED AS ORC;
```

Hash function: `bucket_id = hash(district) % 8`

Partitioning + Bucketing: Visual Summary



- **Partition pruning**: `WHERE year=2024` reads only middle branch
- **Bucket pruning**: `WHERE district=5` reads only 1 of 3 files

Hive vs. MySQL: Side-by-Side

Feature	MySQL / PostgreSQL	Apache Hive
Data size	GBs (single server)	PBs (distributed)
Latency	Milliseconds	Seconds to minutes
Schema	Schema-on-write	Schema-on-read
Updates	Full CRUD support	Limited (ACID optional)
Storage	Own disk format	HDFS (ORC, Parquet, CSV)
Indexes	B-tree, Hash	Partition pruning
Transactions	Full ACID	Limited ACID support
Use case	OLTP (web apps)	OLAP (analytics)

Key Takeaway

Hive is designed for **analytical workloads** (OLAP) on **massive datasets**, not for transactional workloads (OLTP).

Execution Engines: MapReduce vs. Tez vs. Spark

MapReduce

- Original engine
- Disk-based between stages
- Reliable but **slow**
- Good for batch ETL

Slowest

Apache Tez

- DAG-based execution
- Fewer disk writes
- **Default** since Hive 2.0
- 3–10x faster than MR

Faster

Apache Spark

- In-memory processing
- Best for iterative queries
- Uses Spark SQL engine
- Most flexible

Fastest

Setting the Engine

`SET hive.execution.engine = tez;` (or `mr`, `spark`)

Key Takeaways

1. **Hive** = SQL interface to Hadoop / HDFS (not a database!)
2. **Schema-on-read**: Load data first, define schema when querying
3. **HiveQL**: SQL-like language compiled to distributed jobs
4. **Metastore**: Central catalog of table definitions
5. **Managed vs. External**: Know when each is appropriate
6. **Partitioning**: Physical directories for fast filtering
7. **Bucketing**: Hash-based splits for joins and sampling
8. **ORC / Parquet**: Columnar formats for fast analytics

Remember

Hive	SQL on Big Data
HDFS	Where the data lives
Metastore	Where the schema lives

Next Session: Hands-On HiveQL

Week 4B: HiveQL Queries in Practice

- Write HiveQL DDL (CREATE, ALTER, DROP)
- Load data and write queries
- Apply partitioning and performance techniques

Pre-Class Preparation

Get ready for Week 4B:

- Review SQL basics
- Access the Hive environment on the cluster