Big Data Processing and Applications – Lecture 6

Ioana Ciuciu

ioana.ciuciu@ubbcluj.ro

Babes-Bolyai University, Cluj-Napoca

7 Nov. 2025

Course structure

Date	Course/Week	Title
03.10.2025	1	Introduction to Data Science and Big Data (Part 1)
10.10.2025	2	Introduction to Data Science and Big Data (Part 2)
17.10.2025	3	Industrial standards for data mining projects. Big data case studies from industry – invited lecture from Bosch
24.10.2025	4	Data systems and the lambda architecture for big data
31.10.2025	5	Lambda architecture: batch layer
07.11.2025	6	Lambda architecture: serving layer
	7	Lambda architecture: speed layer
	8	NoSQL Solutions for Big Data – invited lecturer from UBB
	9	Data Ingestion
	10	Introduction to SPARK – invited lecture from Bosch
	11	Data visualization
	12	Presentation research essays
	13	Presentation research essays + Project Evaluation during Seminar
	14	Presentation research essays + Project Evaluation during Seminar

Slight modifications in the structure are possible

Semester Project

- Team-based (2-5 students with precise roles)
- Multidisciplinary: 3 CS Master Programes (HPC, SI, DS) + Master in Bioinformatics
- Real use cases: collaboration with local IT industry - TBA
- High degree of autonomy in selecting the topic and proposing the solution
- Implementation prototype
- Prototype demo & evaluation – student workshop (last seminar – weeks 13 & 14)
- *Best projects – disseminated in various events (TBD), scientifically disseminated (workshops/conferences/journals) AND/ OR possibility for a dissertation thesis*

Evaluation

- The final grade will be computed as follows:
  - 50% semester project (must be >= 5)
  - 50% research presentation or written exam (must be >= 5)

- Semester project
  - Details available on the course team
    - MS Team: **Big Data Master - fall 2025**
    - Access code: **j1exenq**

Big Data Master - fall 2025

Agenda

Hadoop Ecosystem

Lambda architecture: Serving Layer (Chapter 10)

- Requirements for the serving layer database
- Performance metrics for the serving layer
- Serving layer: the solution to the normalization/denormalization problem
- Solutions for the serving layer: Apache Hive

Additional support material: **Nathan Matz, James Warren**, Big Data: *Principles and Best Practices of Scalable Real Time Data Systems*

Hadoop Ecosystem

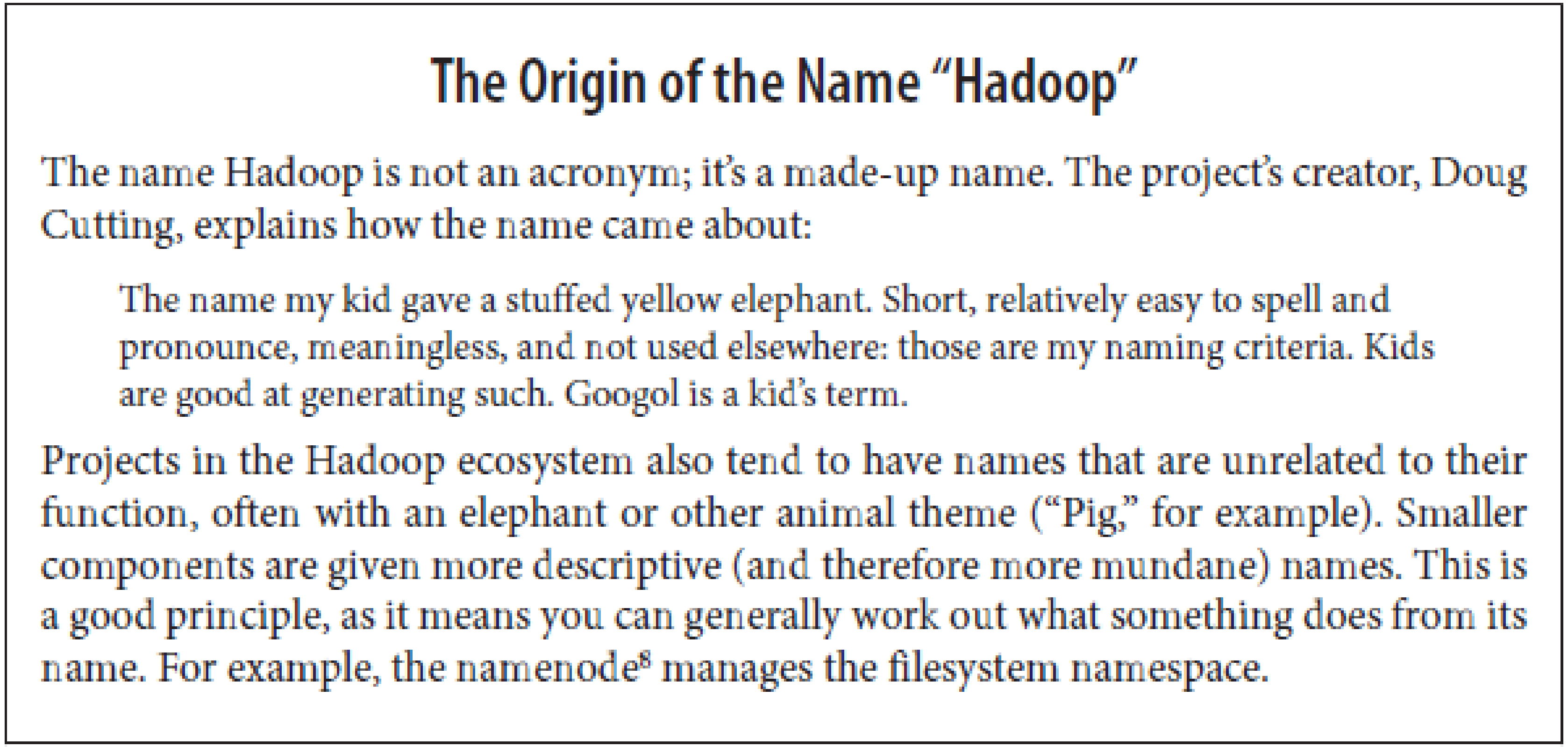
Hadoop Ecosystem: Introduction

- Apache Hadoop is an implementation of Google’s MapReduce framework
- **A tool that can be used for the Batch Layer of the Lambda Architecture**
- It is an open source project hosted by Apache Software Foundation
- It is implemented in Java with bindings for many other languages
- Commercial support and vendor specific additions are available
- Hadoop simplifies the development of data processing tasks and runs on commodity hardware (standardized servers that can be bought from any vendor)

In a nutshell, Hadoop provides a reliable, scalable platform for storage and analysis. Moreover, because it runs on commodity hardware and is open source, Hadoop is affordable.

Hadoop Ecosystem: a brief history

- Hadoop was created by Doug Cutting, the creator of Apache Lucene (the widely used text search library)
- Hadoop has its origins in Apache Nutch, an open web search engine (part of the Lucene project)



The Origin of the Name "Hadoop"

The name Hadoop is not an acronym; it's a made-up name. The project's creator, Doug Cutting, explains how the name came about:

> The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term.

Projects in the Hadoop ecosystem also tend to have names that are unrelated to their function, often with an elephant or other animal theme ("Pig," for example). Smaller components are given more descriptive (and therefore more mundane) names. This is a good principle, as it means you can generally work out what something does from its name. For example, the namenode[8] manages the filesystem namespace.

Source: Hadoop: the Definitive Guide, by Tom White

Hadoop Ecosystem: a brief history

2002: the start of Nutch; a working crawler and search system quickly emerged

2003: a paper describing the architecture of Google's distributed filesystem (GFS) was published

2004: Nutch's developers started to write an open source implementation, the Nutch Distributed Filesystem (NDFS)

2004: Google published the paper that introduced MapReduce[1]

2005: all major Nutch algorithms had been ported to run using MapReduce and NDFS

2006: an independent subproject of Lucene was started, called Hadoop

At around the same time, Doug Cutting joined Yahoo!, providing a dedicated team and the resources to turn Hadoop into a system that ran at web scale, demonstrated in

2008: Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster

[1] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," December 2004.

Hadoop Ecosystem: a brief history

2008, January: Hadoop was made its own top-level project at Apache

Many other companies were using Hadoop during this time, such as Last.fm, Facebook, and the *New York Times*

2008, April: Hadoop broke a world record to become the fastest system to sort an entire terabyte of data on a 910-node cluster

Hadoop sorted 1 terabyte in 209 seconds (just under 3.5 minutes), beating the previous year’s winner of 297 seconds

2008, November: Google reported that its MapReduce implementation sorted 1 terabyte in 68 seconds

2009, April: a team at Yahoo! has used Hadoop to sort 1 terabyte in 62 seconds

Since then, the trend has been to sort even larger volumes of data at ever faster rates

Hadoop Ecosystem: a brief history

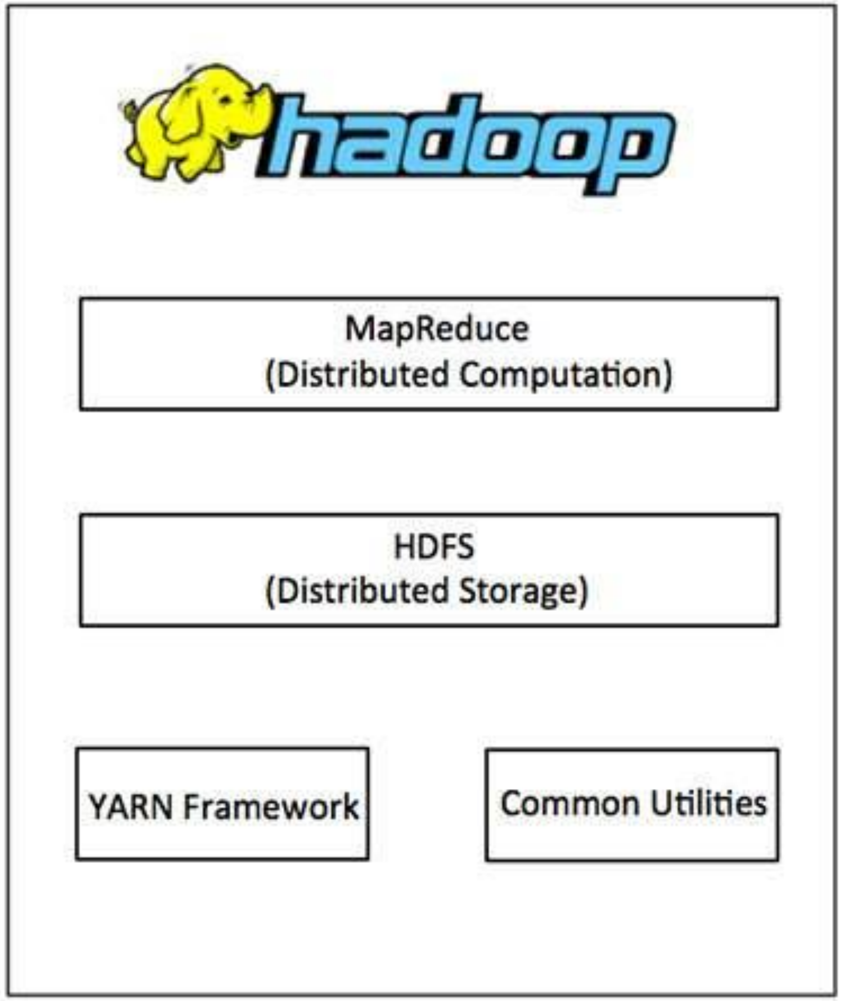
Today: Hadoop is widely used in mainstream enterprises
The industry recognized Hadoop’s role as a general-purpose storage and analysis platform for Big Data

Commercial Hadoop support is available from large vendors, including EMC, IBM, Microsoft and Oracle

Commercially available Hadoop platforms:

- Cloudera (previously also Hortonworks (merged with Cloudera)
- MapR (merged with HP))
- Amazon Web Services
- IBM
- Microsoft HDInsight
- Intel Distribution for Apache Hadoop
- …

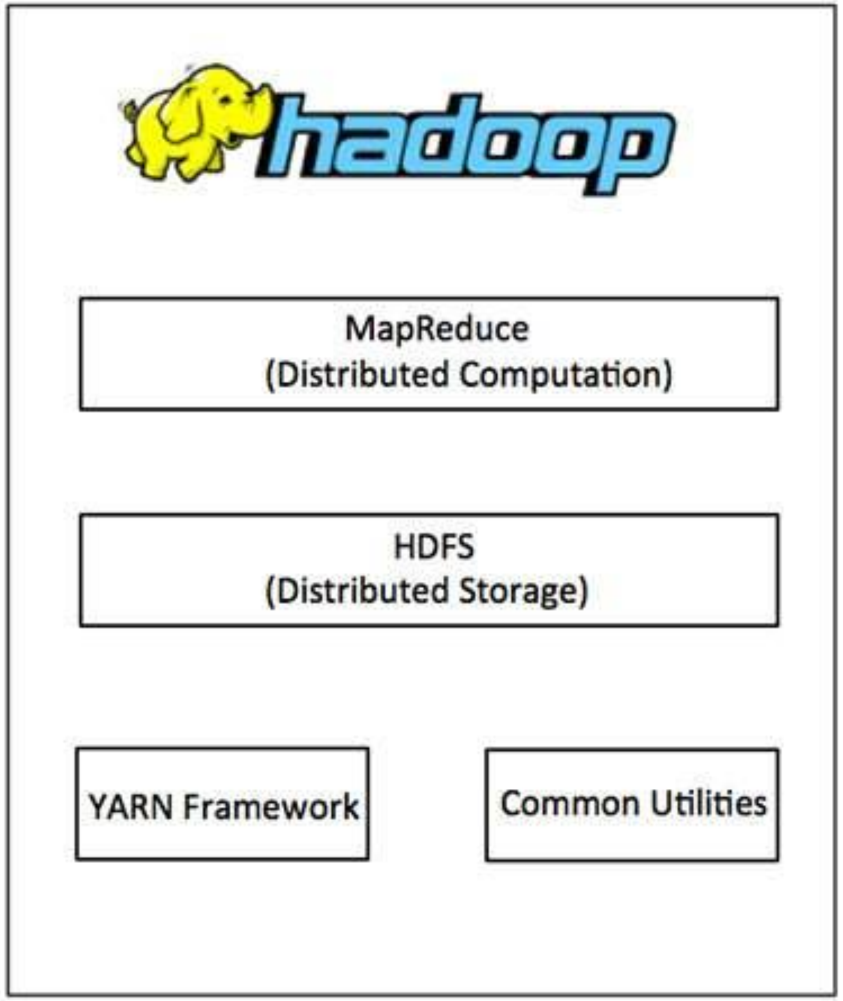
Hadoop Ecosystem: Architecture



Hadoop Ecosystem: Architecture

Hadoop Common − Contains Java libraries and utilities required by other Hadoop modules.

Hadoop YARN − Is a framework for job scheduling and cluster resource management.

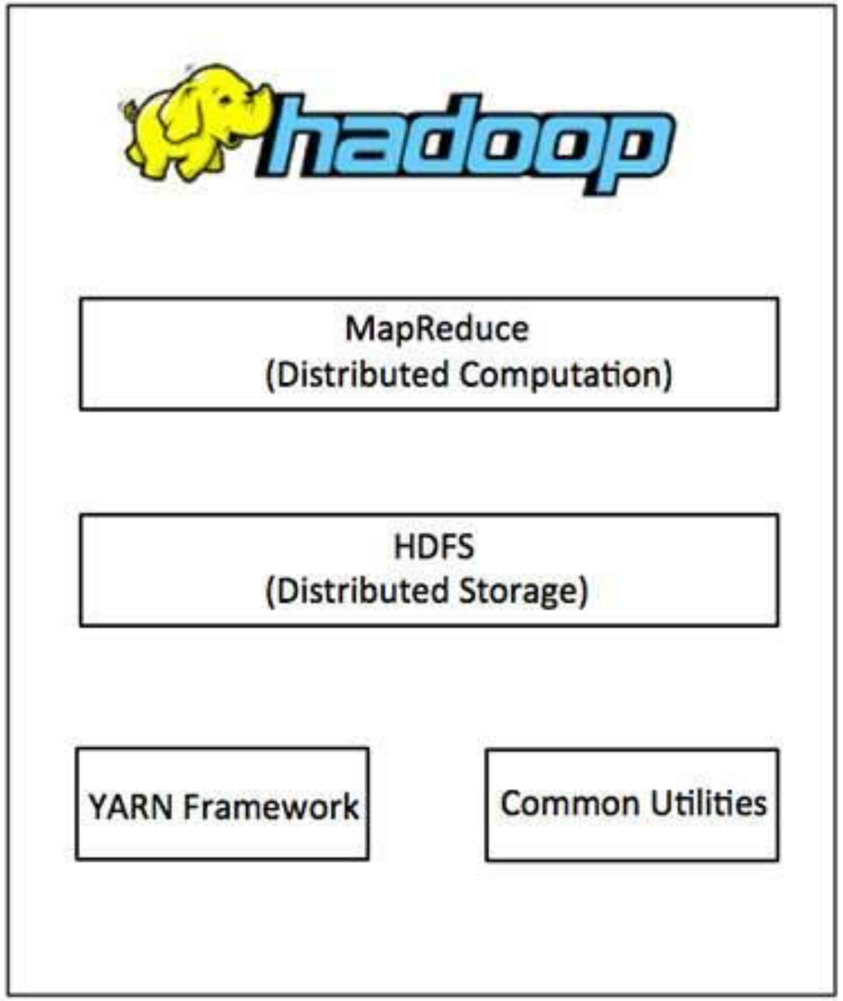


Hadoop Ecosystem: Architecture

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many similarities with existing distributed file systems.

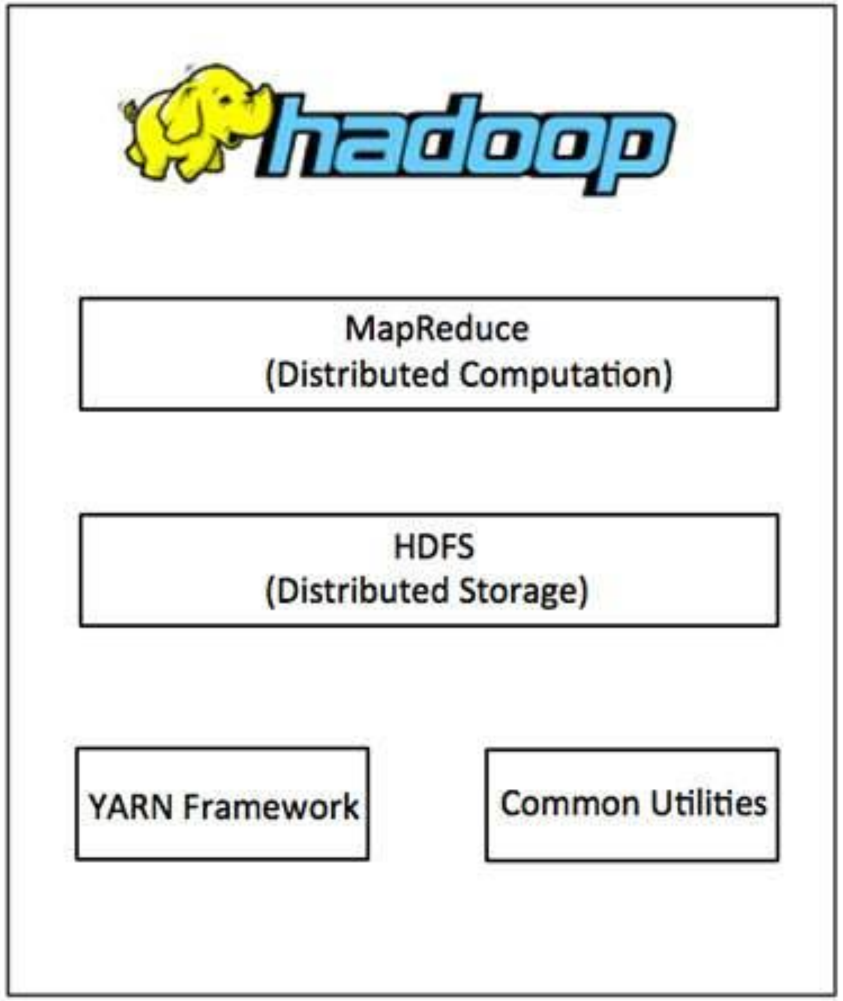
Key features for HDFS:

- It is highly fault-tolerant and is designed to be deployed on low-cost hardware.
- It provides high throughput access to application data and is suitable for applications having large datasets.

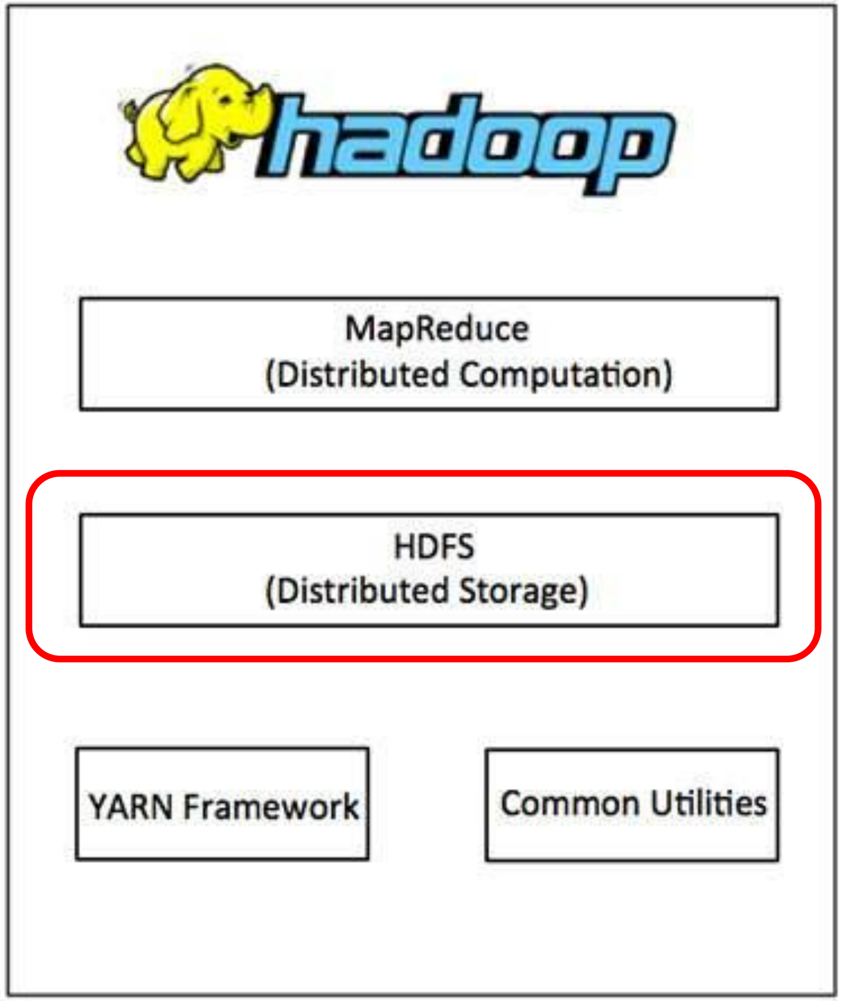


Hadoop Ecosystem: Architecture

MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte data-sets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

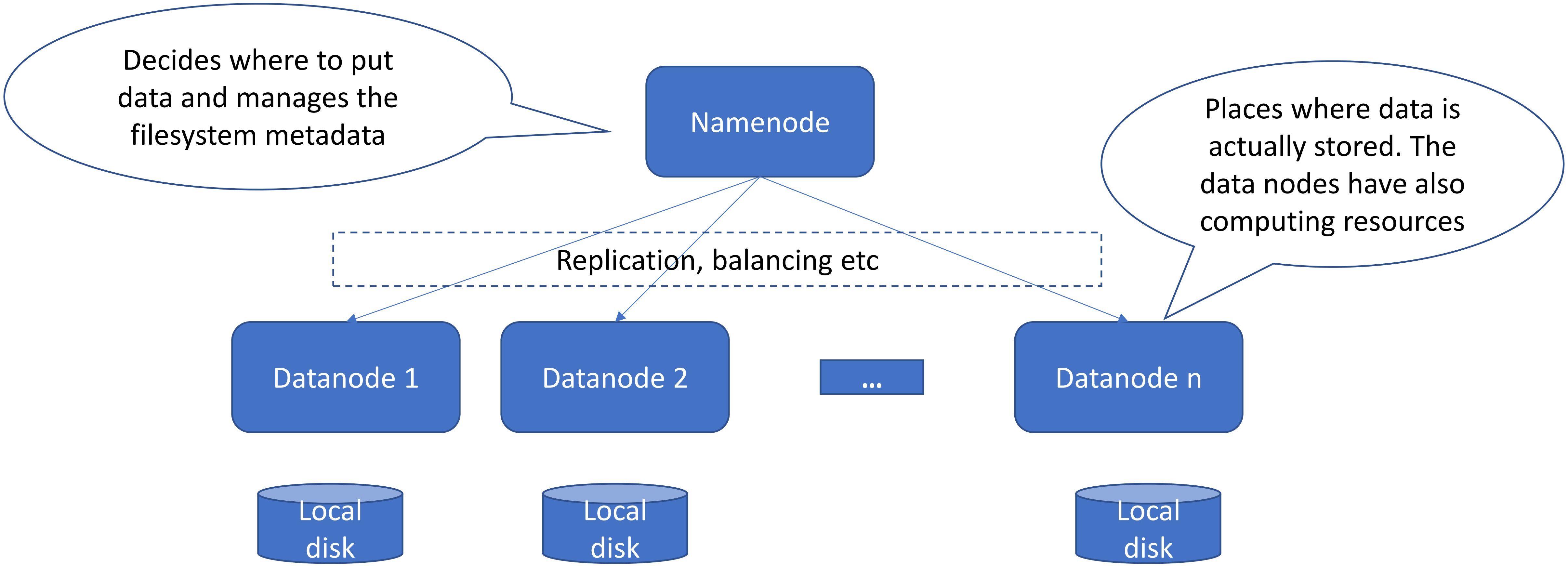


Hadoop Ecosystem: Hadoop Distributed File System (HDFS)



Hadoop Ecosystem: Hadoop Distributed File System (HDFS)

HDFS: architecture



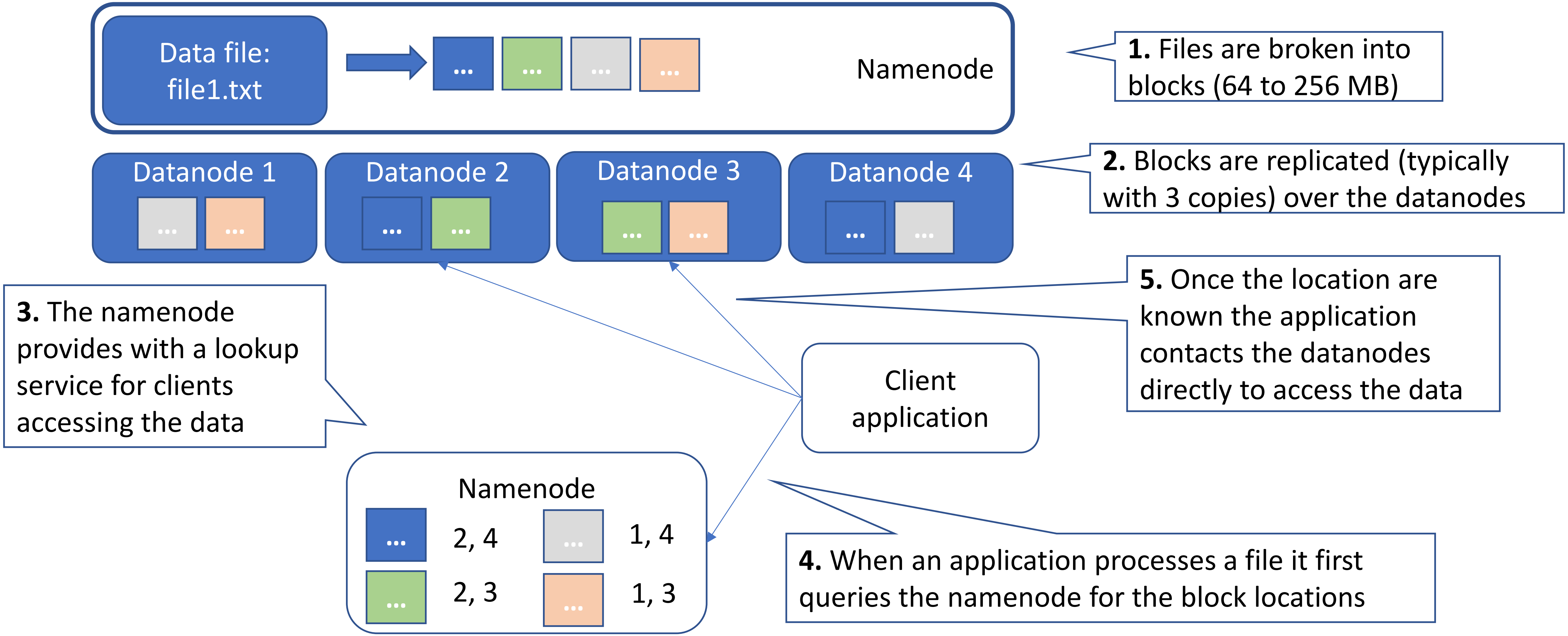
Decides where to put data and manages the filesystem metadata

Places where data is actually stored. The data nodes have also computing resources

Replication, balancing etc

Hadoop Ecosystem: Hadoop Distributed File System (HDFS)

HDFS: how does it work?



1. Files are broken into blocks (64 to 256 MB)

2. Blocks are replicated (typically with 3 copies) over the datanodes

3. The namenode provides with a lookup service for clients accessing the data

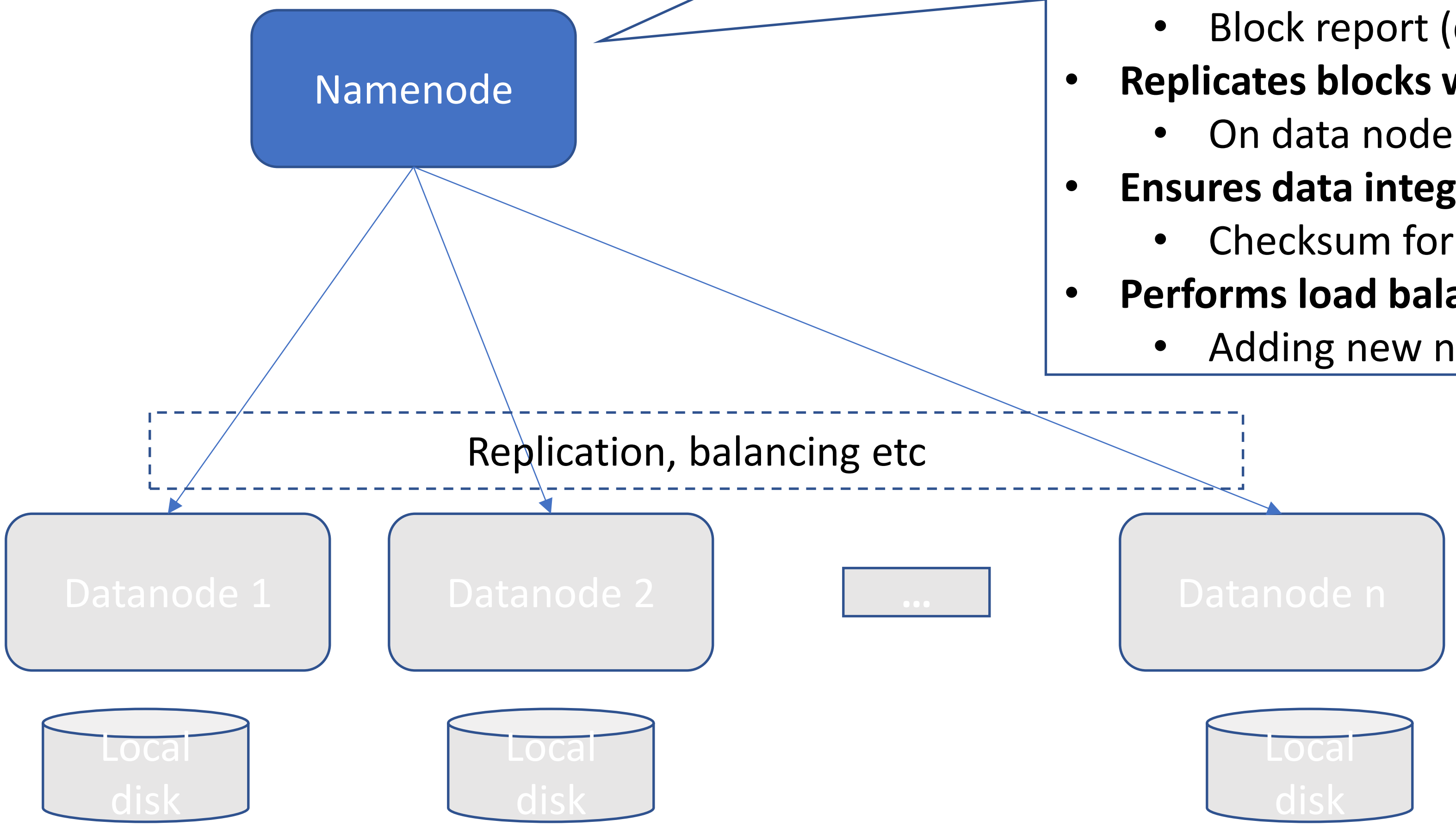
4. When an application processes a file it first queries the namenode for the block locations

5. Once the location are known the application contacts the datanodes directly to access the data

Hadoop Ecosystem: Hadoop Distributed File System (HDFS)

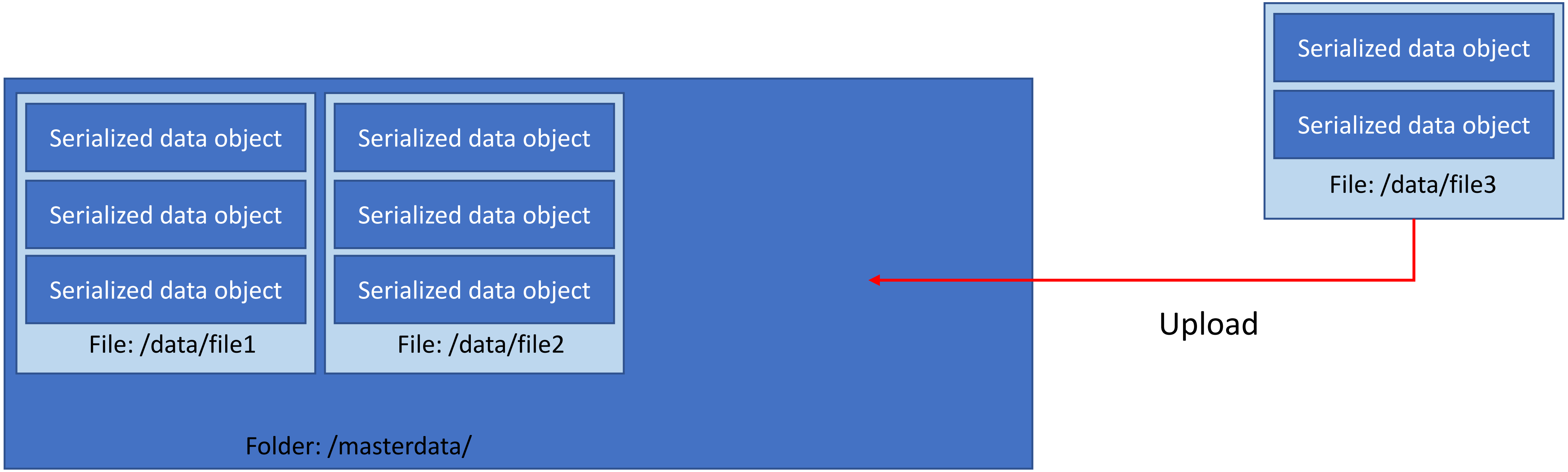
Name node functionalities

- **Manages File System** – maps files to blocks and blocks to data nodes
- **Maintains the status of data nodes**
  - Heartbeat (data nodes send heartbeats at regular intervals; if heartbeat is not received, the node is declared dead)
  - Block report (data nodes send list of blocks on it)
- **Replicates blocks whenever necessary**
  - On data node failure, disk failure or on block corruption
- **Ensures data integrity**
  - Checksum for each block
- **Performs load balancing**
  - Adding new nodes, decommissioning



Hadoop Ecosystem: Hadoop Distributed File System (HDFS)

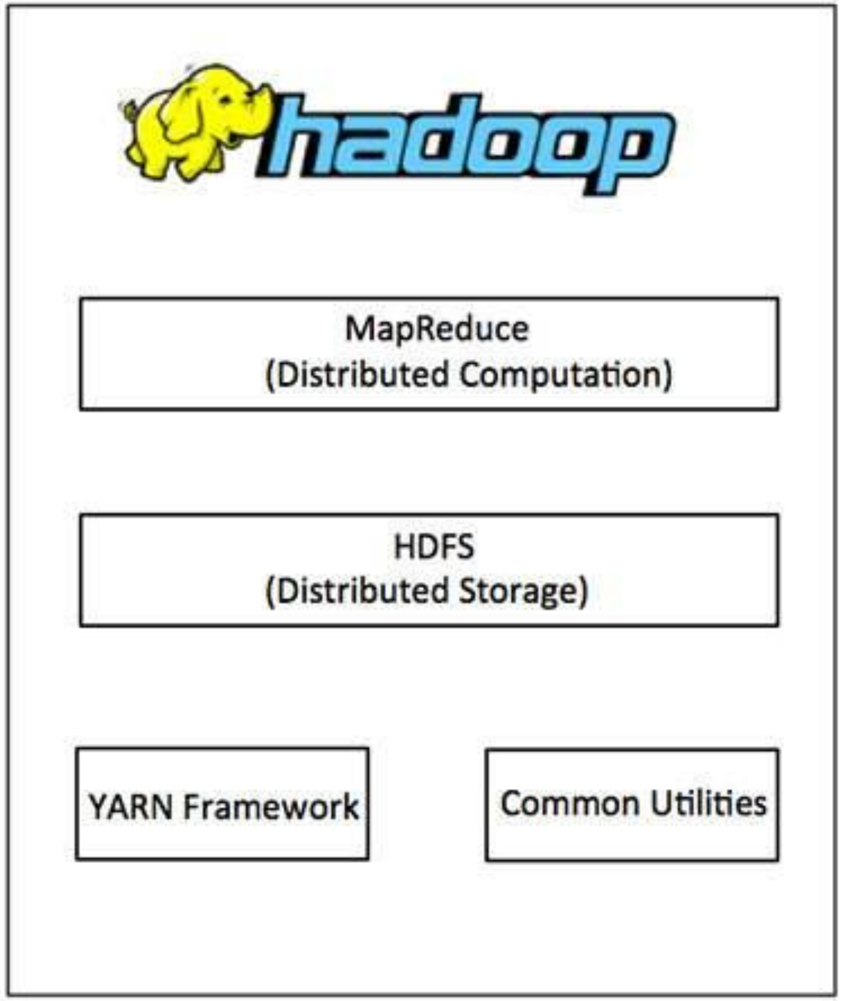
Storing the master dataset



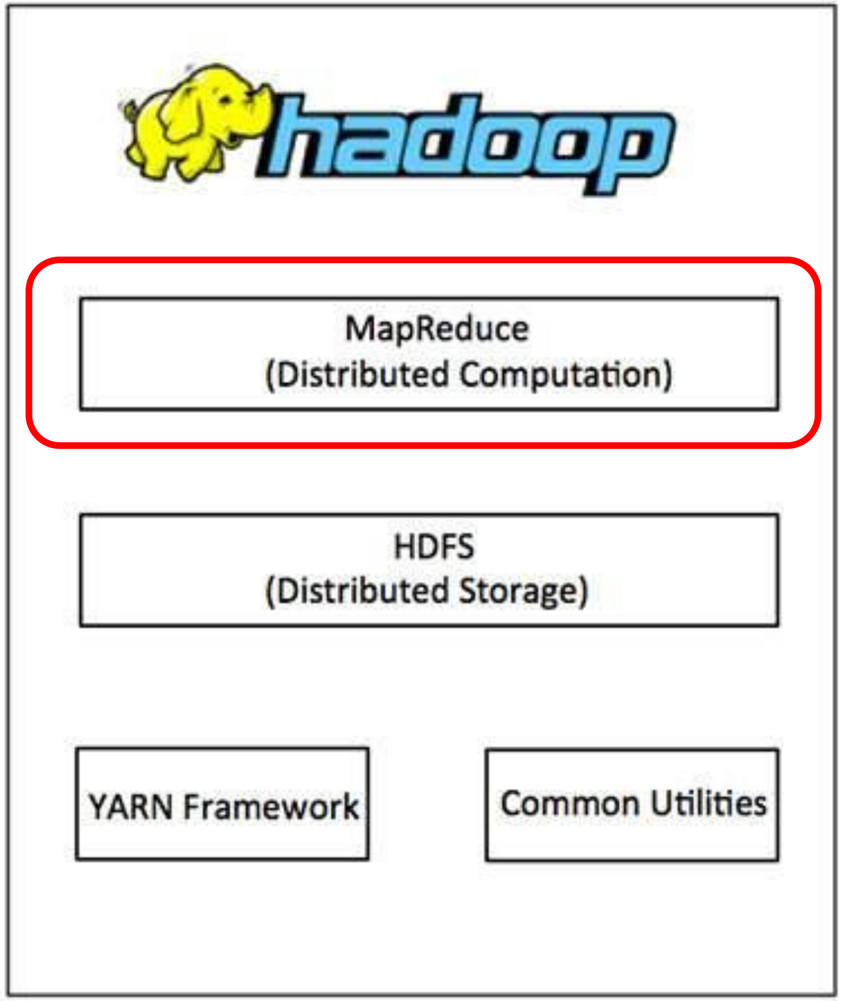
Hadoop Ecosystem: Hadoop Distributed File System (HDFS)

Key features for HDFS:

- It is highly fault-tolerant and is designed to be deployed on low-cost hardware.
- It is scalable on demand.
- Files are distributed in large blocks for efficient read and parallel access
- It provides high throughput access to application data and is suitable for applications having large datasets.

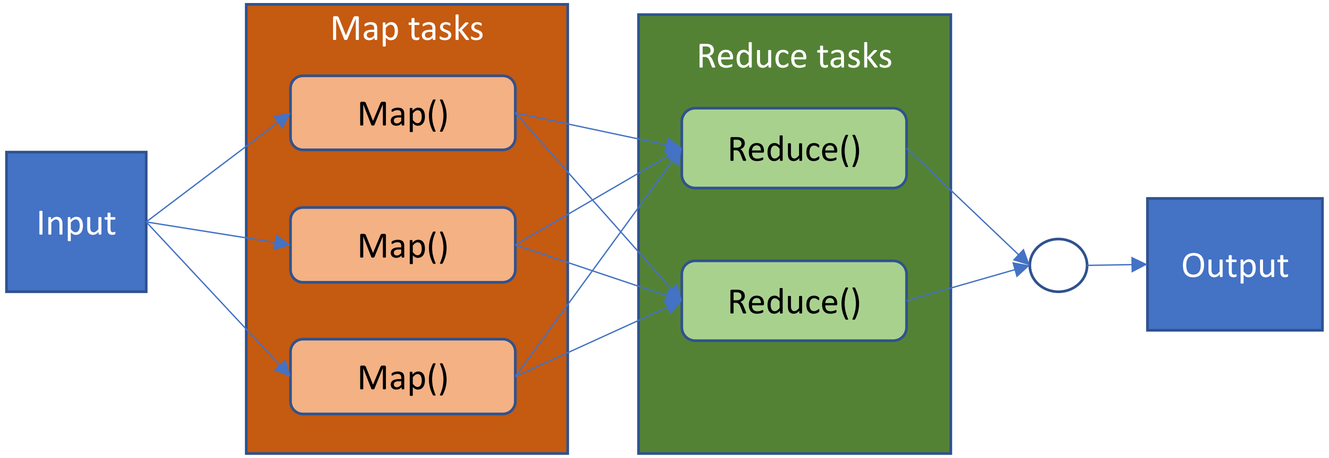


Hadoop Ecosystem: MapReduce



Hadoop Ecosystem: MapReduce

***MapReduce* how it works?**

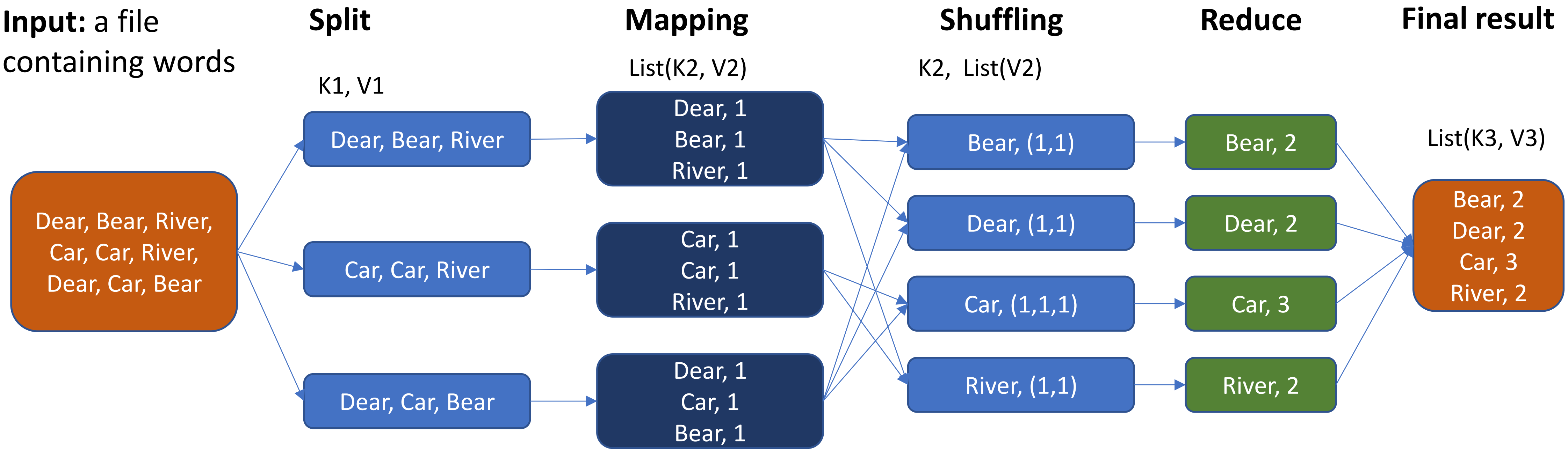


1. The first is the *Map* job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
2. The output of a Mapper or map job (key-value pairs) is input to the Reducer.
3. The reducer receives the key-value pair from multiple map jobs.
4. Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

Hadoop Ecosystem: MapReduce

***MapReduce* how it works?**

Example: word count



Input: a file containing words

Dear, Bear, River, Car, Car, River, Dear, Car, Bear

Split

K1, V1

Dear, Bear, River

Car, Car, River

Dear, Car, Bear

Mapping

List(K2, V2)

Dear, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Dear, 1
Car, 1
Bear, 1

Shuffling

K2, List(V2)

Bear, (1,1)

Dear, (1,1)

Car, (1,1,1)

River, (1,1)

Reduce

Bear, 2

Dear, 2

Car, 3

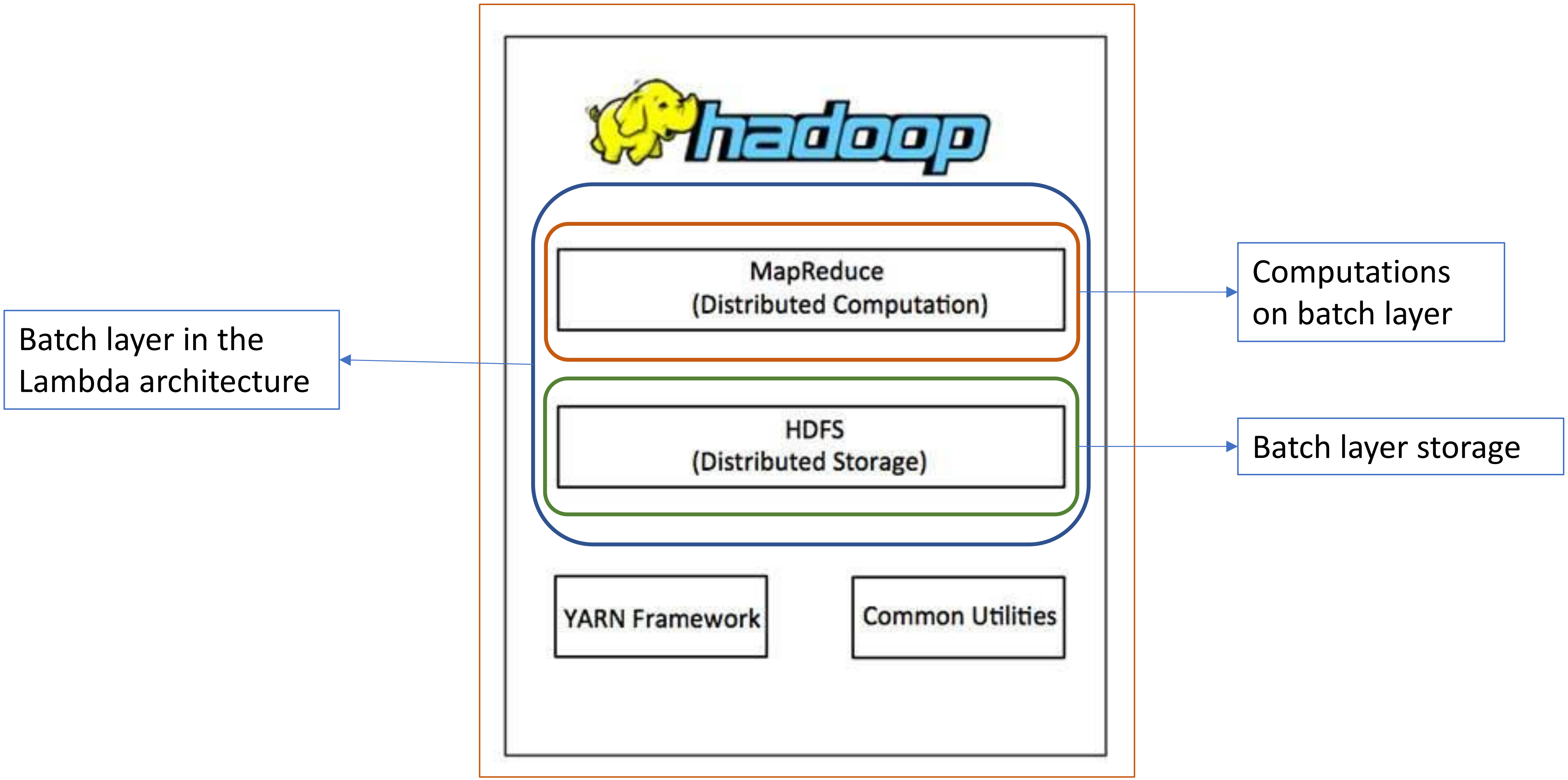
River, 2

Final result

List(K3, V3)

Bear, 2
Dear, 2
Car, 3
River, 2

Hadoop Ecosystem: Summary



hadoop

MapReduce
(Distributed Computation)

HDFS
(Distributed Storage)

YARN Framework

Common Utilities

Batch layer in the
Lambda architecture

Computations
on batch layer

Batch layer storage

Hadoop Ecosystem: further readings

1. Official Hadoop Website: https://hadoop.apache.org/

2. Hadoop: The Definitive Guide. Tom White

3. Hadoop in practice. Alex Holmes, 2nd edition

Hadoop Ecosystem – take home

- Implements the batch layer

- Two main component for the main functionalities of the batch layer:
  - HDFS – storage on the batch layer
  - MapReduce – computations on the batch layer

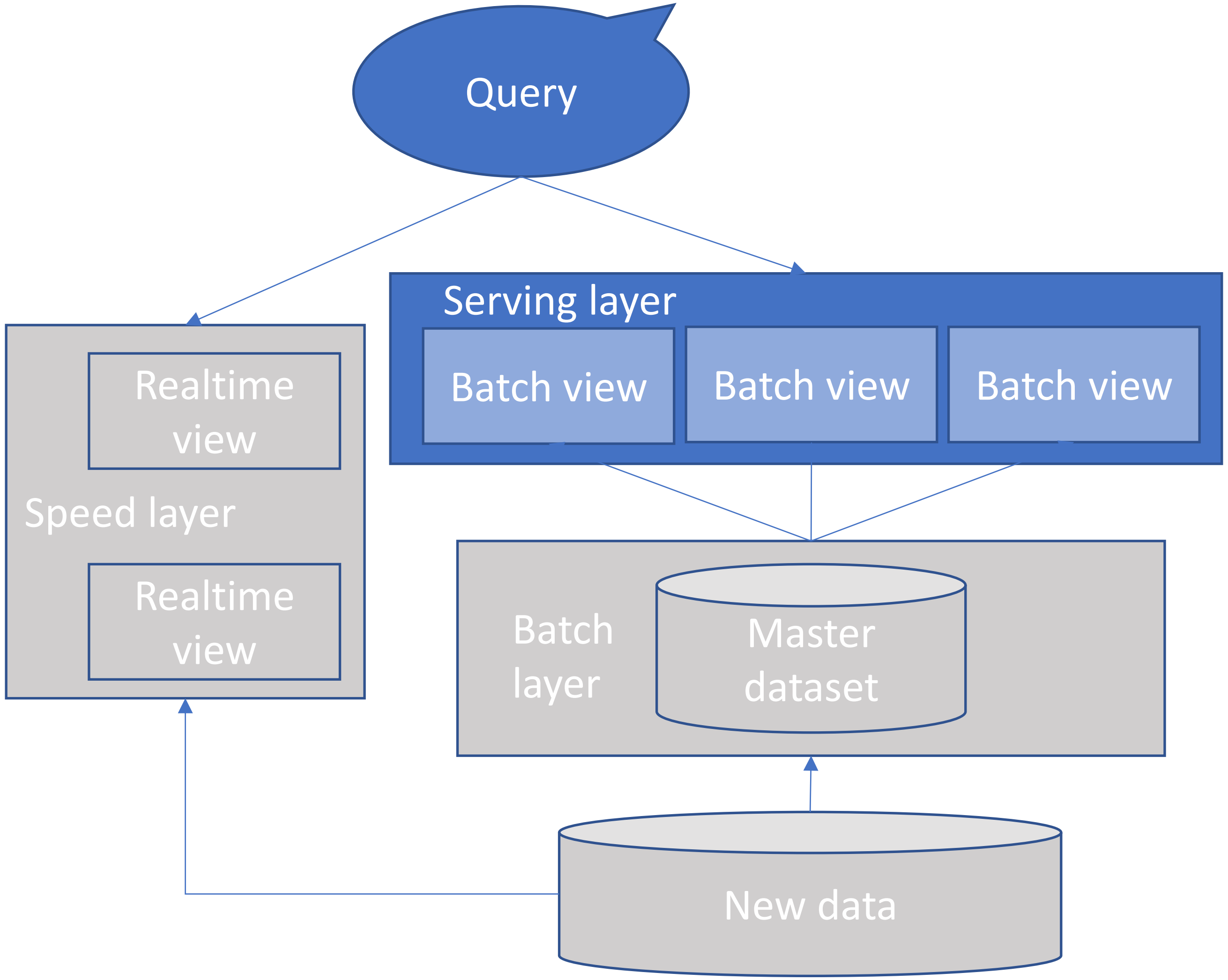
Lambda architecture: serving layer

Lambda architecture: serving layer

Batch layer: implements the equation below

batch view = function (all data)

Serving layer: indexes the views and provides interfaces so that the precomputed data can be easily queried



Lambda architecture: serving layer

A serving layer database **does not need to support random writes**; as a consequence such databases are extremely simple

Lambda architecture: serving layer

Performance metrics for serving layer database

Latency – the time required to answer a single query

Throughput – the number of queries that can be served within a given period of time

Note: both latency and the throughput are influenced by the strategy used for indexing

Lambda architecture: serving layer

Performance metrics for serving layer database

Remark

For a given database with known limits for throughput and latency, the indexing strategy plays the major role on the overall performance of the system.

The main question to be answered when designing a data system is what indexing strategy to be used such that the system can serve as many as possible requests with a low latency.

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly Bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [Sensor, hour] pair as keys and the mean temperature as values

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

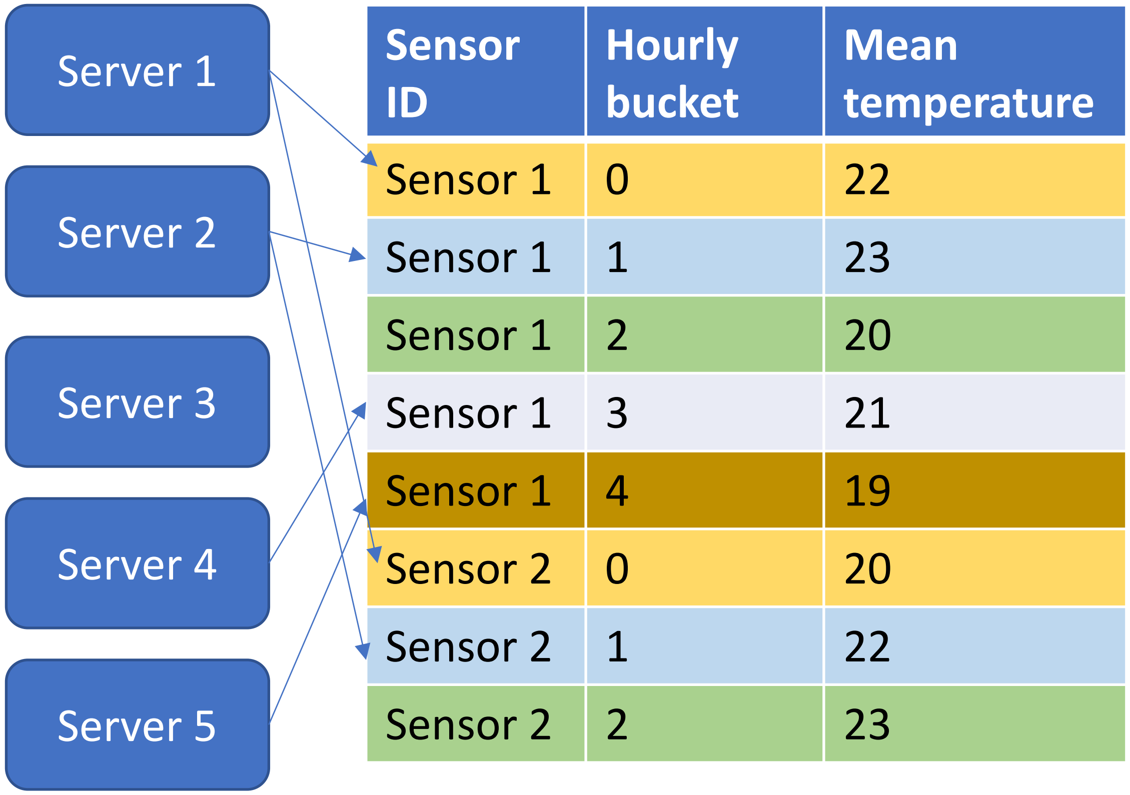
Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

The index would be partitioned by key so mean temperature values from the same sensor would be on different partitions

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours



Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

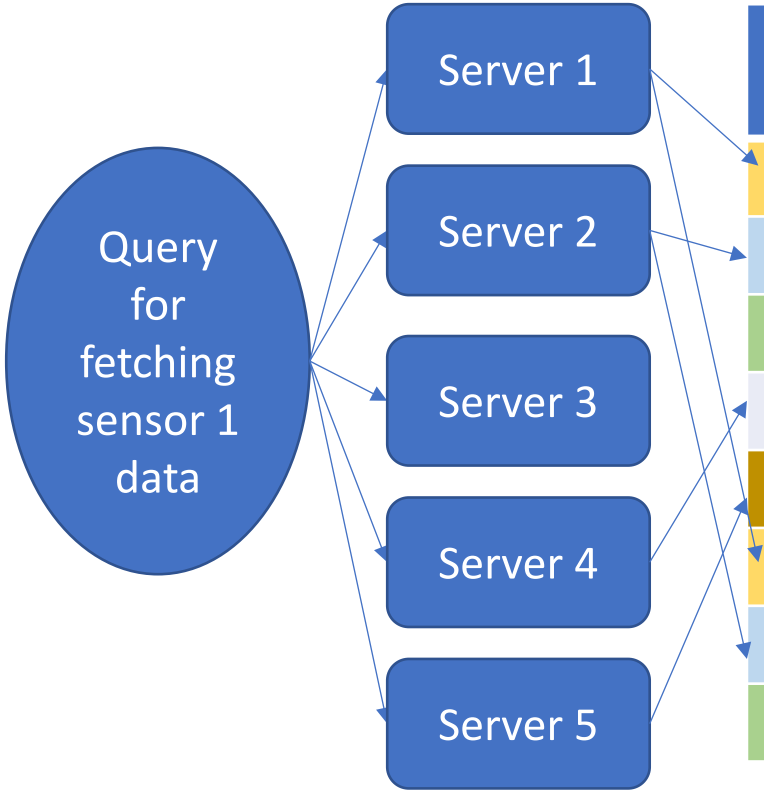
The index would be partitioned by key so mean temperature values from the same sensor would be on different partitions

Different partitions would be located on different servers

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours



Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

The index would be partitioned by key so mean temperature values from the same sensor would be on different partitions

Different partitions would be located on different servers

Retrieving the range of hours for a sensor would involve fetching data from multiple servers in your serving layer

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Latency is significantly high

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Latency is significantly high because:

1. You need to query multiple servers to fetch your data

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Latency is significantly high because:

1. You need to query multiple servers to fetch your data
2. The response time of your servers may vary; e.g. one server could be much more loaded than others

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Latency is significantly high because:

1. You need to query multiple servers to fetch your data
2. The response time of your servers may vary; e.g. one server could be much more loaded than others
3. The overall query response is limited by the speed of the slowest server

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Latency is significantly high because:

1. You need to query multiple servers to fetch your data
2. The response time of your servers may vary; e.g. one server could be much more loaded than others
3. The overall query response is limited by the speed of the slowest server
4. The more servers a query touches, the higher the overall latency of the query since the probability that at least one server will be slow increases

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor because:

1. Retrieving data for a single key requires disk seek

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor because:

1. Retrieving data for a single key requires disk seek
2. A single query may involve tens of keys

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor because:

1. Retrieving data for a single key requires disk seek
2. A single query may involve tens of keys
3. Disk seeks are expensive for traditional hard drives

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor because:

1. Retrieving data for a single key requires multiple disk seek
2. A single query may involve tens of keys
3. Disk seeks are expensive for traditional hard drives
4. The number of disks on a cluster is finite -> there is a hard limit on the number of disk seeks that can be served per second

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours

Sensor ID	Hourly bucket	Mean temperature
Sensor 1	0	22
Sensor 1	1	23
Sensor 1	2	20
Sensor 1	3	21
Sensor 1	4	19
Sensor 2	0	20
Sensor 2	1	22
Sensor 2	2	23

Indexing strategy 1: use a key/value strategy with [sensor, hour] pair as keys and the mean temperature as values

Throughput is poor: example

Suppose that

- On average a query fetches 20 keys
- The cluster has 100 disks
- Each disk can perform 500 seeks per second

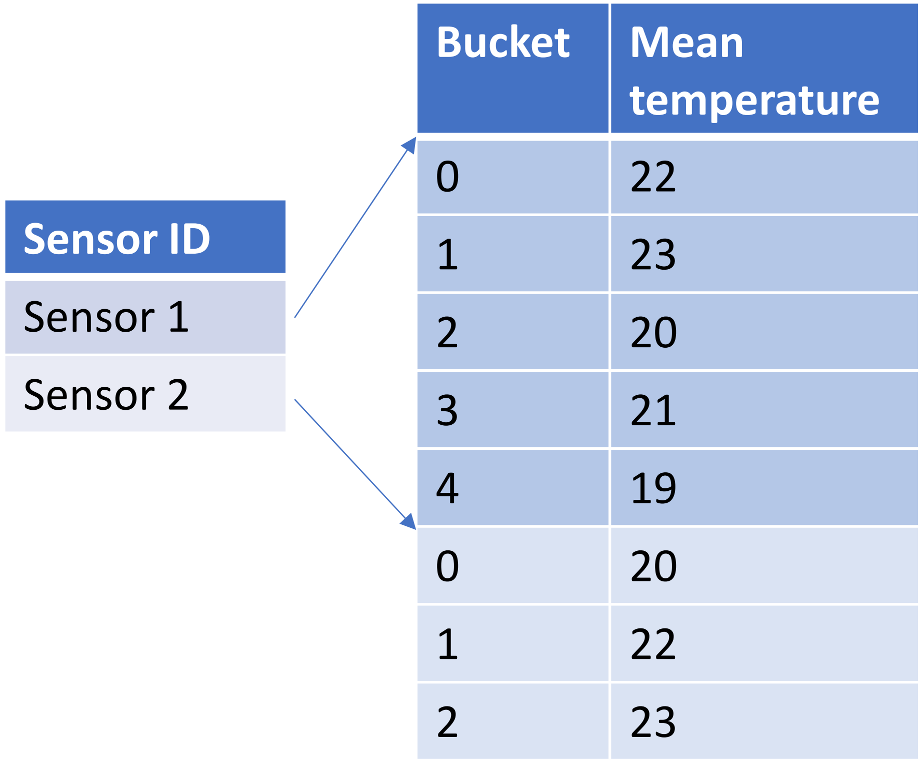
- How many queries can the cluster serve?

  - 100x500/20 = 2500

Lambda architecture: serving layer

Performance metrics for serving layer database

Example: you own a network of sensors and you want to serve the mean temperature for each hour given a particular sensor ID and a range of hours



Sensor ID
Sensor 1
Sensor 2

Bucket	Mean temperature
0	22
1	23
2	20
3	21
4	19
0	20
1	22
2	23

Indexing strategy 2: collocate the mean temperature values for the same sensor on the same partition and store it sequentially

Advantages:

1. Fetching the data will require a single seek and scan instead of multiple seeks
2. A single server will be contacted per query so the probability to have a high latency decreases

Lambda Architecture allows you to tailor the serving layer for the queries it serves to optimize efficiency

Lambda architecture: serving layer

Requirements for the serving layer database

Required	Description
Batch writable	The batch views for the serving layer are produced from scratch. When a new version of the view becomes available, it must be possible to completely swap out the older version with the updated view
Scalable	A serving layer database should be capable to handle views of arbitrary size. As with the DFS and the batch computation framework, this requires it to be distributed across multiple machines
Support random reads	Must support random reads, with indexes providing direct access to small portions of the view. This requirement is necessary to have low latency on queries
Fault-tolerant	Since a serving layer database is distributed, it must also be tolerant to machine failures

Lambda architecture: serving layer

Requirements for the serving layer database

Not Required	Description
Random writes	**Random writes** are required in the Lambda Architecture but on the speed layer
Updates	**Updates** on the serving layer generate new views in their entirety so for the serving layer the updates are completely irrelevant

Serving layer databases are simple →

They are more predictable since they do fewer things

They are less likely to have bugs

Since they contain the huge majority of queryable data, their simplicity is a huge advantage to the robustness of the whole architecture

Lambda architecture: serving layer

The solution to the normalization/denormalization problem

QUIZZ: Which statement is correct?

Data normalization:

- Is a process which consists in storing data in a particular structure such that the latency is minimized
- Is a process which consists in storing data in a particular structure such that the redundancy is maximized
- Is a process which consists in scaling data such that all measurements are in the same range

Lambda architecture: serving layer

The solution to the normalization/denormalization problem

QUIZZ: Which statement is correct?

Data normalization:

- Is a process which consists in storing data in a particular structure such that the latency is minimized
- Is a process which consists in storing data in a particular structure such that the redundancy is maximized
- Is a process which consists in scaling data such that all measurements are in the same range

None

Lambda architecture: serving layer

The solution to the normalization/denormalization problem

QUIZZ: Which statement is correct?

Data normalization:

- Is a process which consists in storing data in a particular structure such that the latency is minimized
- Is a process which consists in storing data in a particular structure such that the redundancy is ~~maximized~~ minimized
- Is a process which consists in scaling data such that all measurements are in the same range

Lambda architecture: serving layer

The solution to the normalization/denormalization problem

The normalization/denormalization problem is a trade-off between redundancy and data consistency

Normalization

- Is implemented by the data model
- **Goal** is **to reduce redundancy**
- Normalization decomposes data into tables to reduce redundancy
- Redundancy in data could lead to inconsistency between different versions of the data stored at different places.
- A fully normalized database allows easy extension of the database structure without changing the existing database. It is possible to add new columns and datatypes to the current database

De-normalization

- Is implemented on a normalized database.
- **Goal** is **to reduce the time required to execute any query on the database**.
- Denormalization combines different data sets to improve time efficiency.
- Denormalization introduces redundancy in the database and therefore could lead to inconsistency.
- A denormalized database allows fast data retrieval

Lambda architecture: serving layer

The solution to the normalization/denormalization problem

The normalization/denormalization problem is a trade-off between redundancy and data consistency

RDMS – data is stored fully normalized

Advantage: redundancy is minimized

Disadvantage: latency is high (queries are very slow)

One can make the queries faster if you store some data redundantly.

Disadvantage: the complexity to keep the data consistent increases significantly

Data consistency must be ensured

Lambda Architecture solves the normalization/denormalization problem

Data consistency is ensured by the batch layer that stores normalized data

Fast data access is ensured by the serving layer that that is completely tailored for the queries it serves

Tools used in the serving layer: Hive

Serving layer tools: Hive

History

What is Hive?

Architecture

How it works?

Data modelling

Data types supported by Hive

Hive vs RDMS

Serving layer tools: Hive

History

Hive was initially developed by Facebook. But why?

1 In 2010 Facebook was already using Hadoop to handle Big Data

2 Hadoop relies on MapReduce for data processing. MapReduce requires users to write code

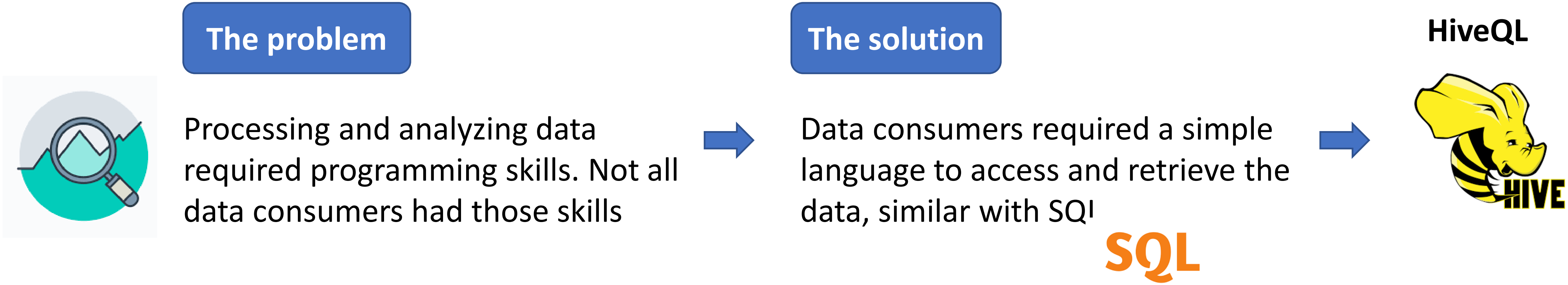
3 But not all users are familiar with writing code. This was a disadvantage for them

4 Users were comfortable with writing queries in SQL

5 Hive was built with the vision to incorporate the concept of tables and columns so that data can be easily retrieved and processed using SQL
Hive was released in 2010

Serving layer tools: Hive

The reason behind Hive



The problem

Processing and analyzing data required programming skills. Not all data consumers had those skills

The solution

Data consumers required a simple language to access and retrieve the data, similar with SQL

Serving layer tools: Hive

What is Hive?

Hive is a data warehousing system used to query and analyze large datasets stored in HDFS or Amazon S3

Hive offers a SQL interface and a query engine

Hive uses its own query language called HiveQL which is similar with SQL

What is NOT … ?

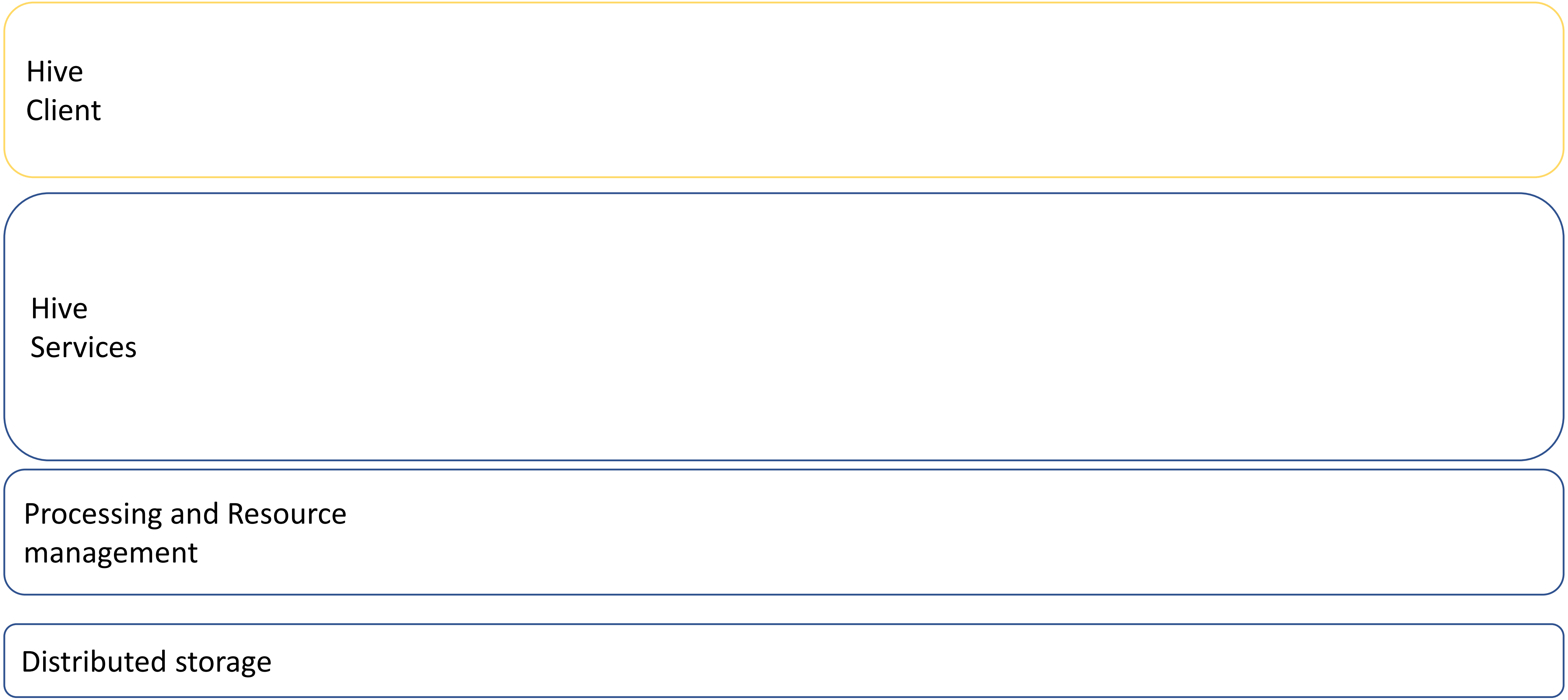
A Relational Database – Hive uses a database to store metadata but the data that Hive processes is stored in HDFS

Designed for On-Line Transaction Processing – Hive runs on Hadoop therefore the latency for Hive queries is high

Not well-suited for real-time queries and row-level updates – Hive is best used for batch processing

Serving layer tools: Hive

Architecture



Hive
Client

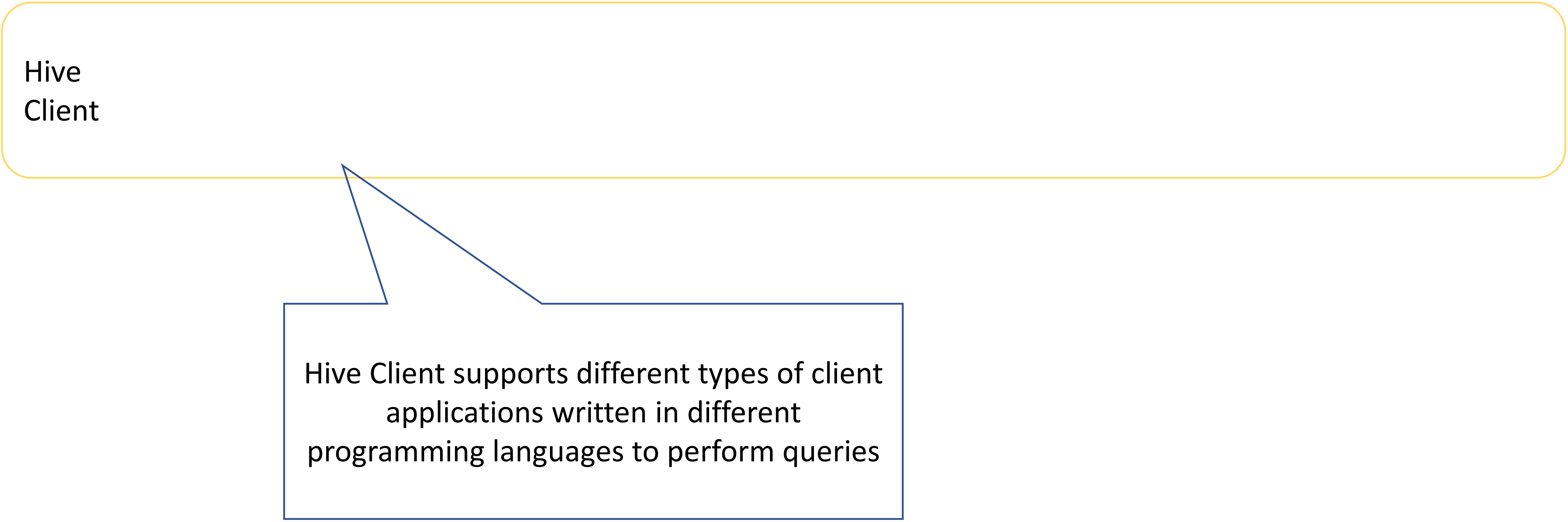
Hive
Services

Processing and Resource management

Distributed storage

Serving layer tools: Hive

Architecture

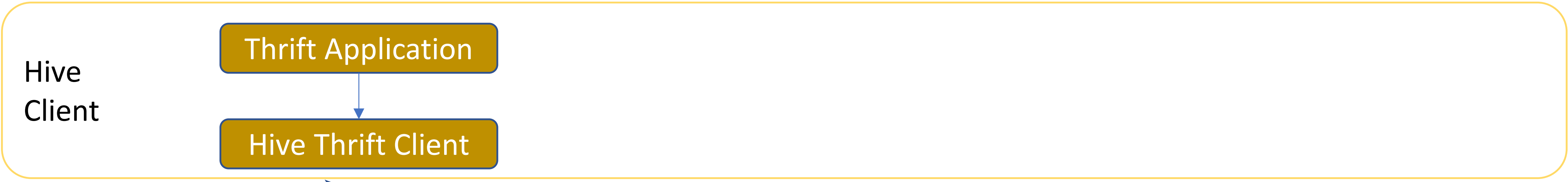


Hive Client

Hive Client supports different types of client applications written in different programming languages to perform queries

Serving layer tools: Hive

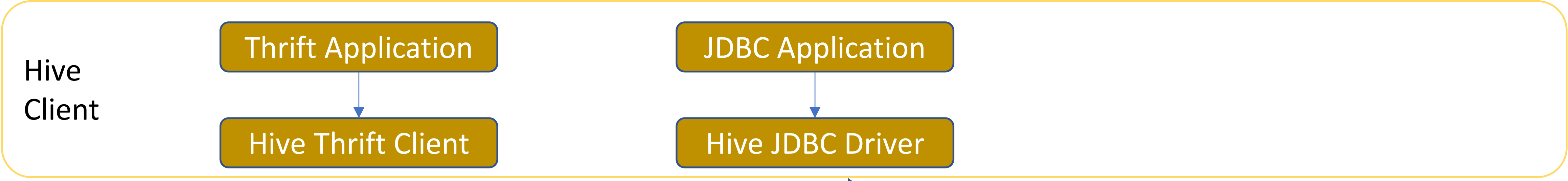
Architecture



Thrift is a software framework used for defining and creating services written in a variety of programming languages and frameworks. Hive Server is based on Thrift so it can serve requests from all programming languages that Thrift supports.

Serving layer tools: Hive

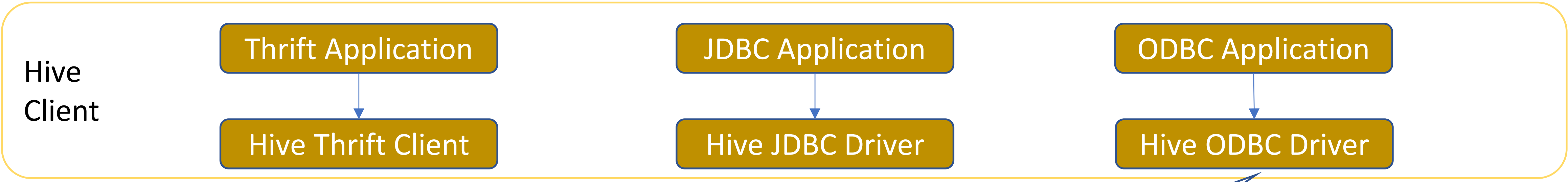
Architecture



JDBC - Java Database Connectivity
JDBC application is connected through the JDBC driver

Serving layer tools: Hive

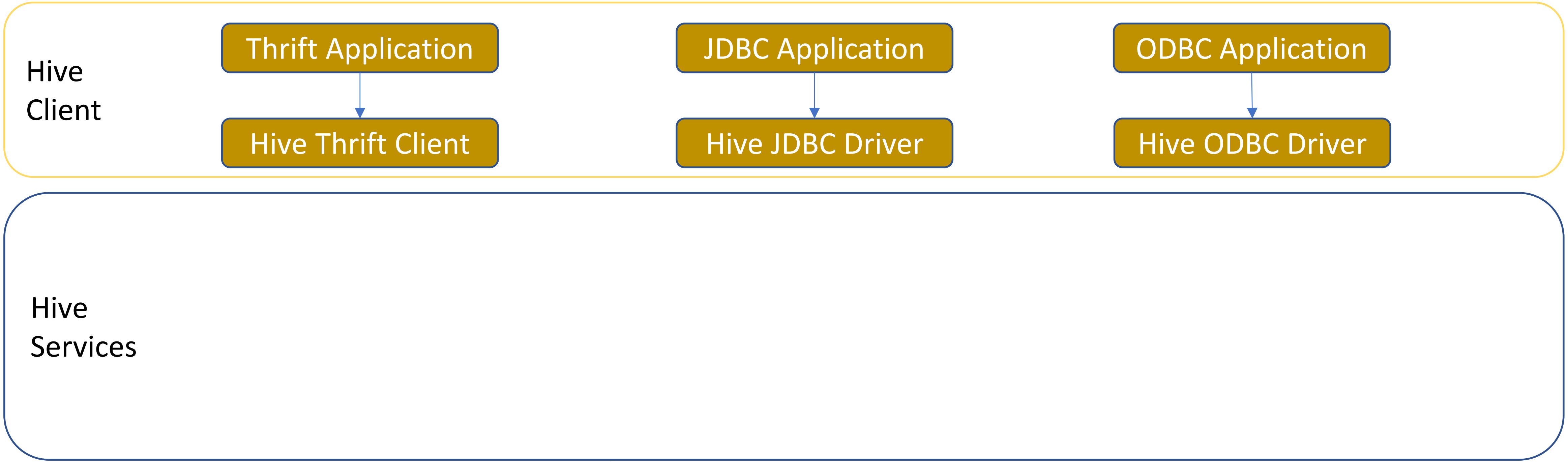
Architecture



ODBC - Open Database Connectivity
ODBC application is connected through the ODBC driver

Serving layer tools: Hive

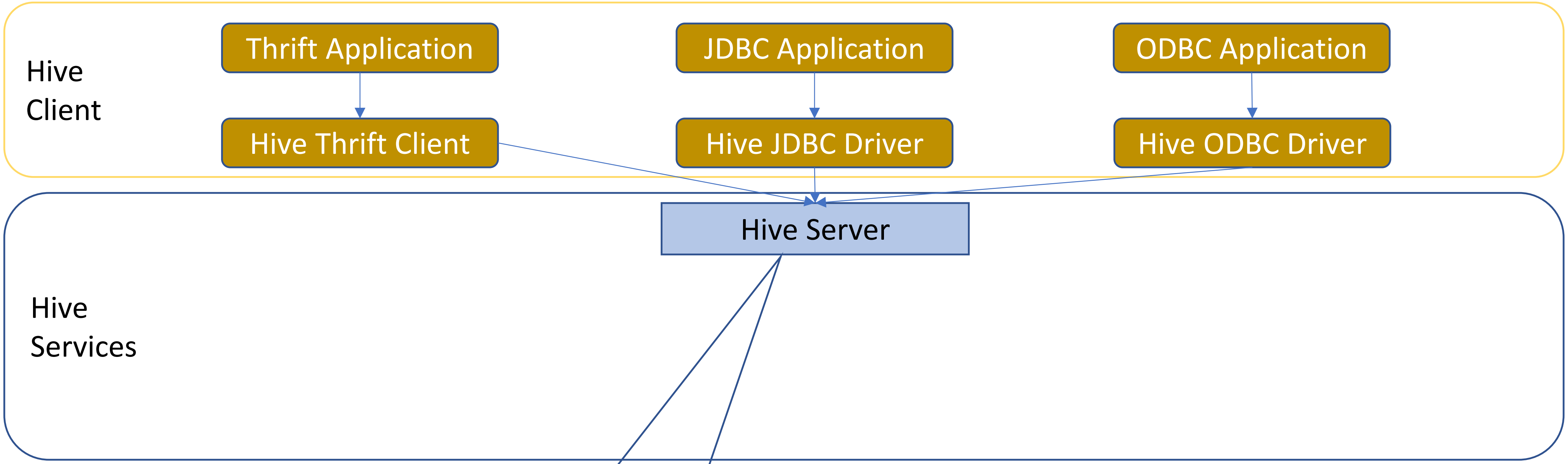
Architecture



Hive supports a number of services

Serving layer tools: Hive

Architecture



Hive Client

Thrift Application

Hive Thrift Client

JDBC Application

Hive JDBC Driver

ODBC Application

Hive ODBC Driver

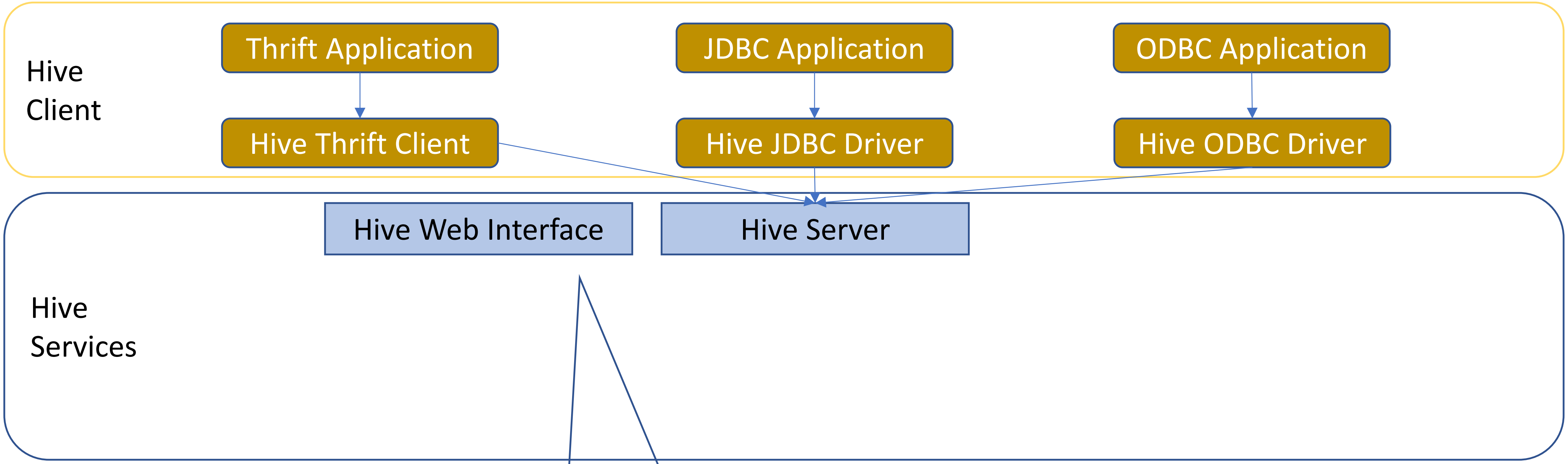
Hive Services

Hive Server

All clients requests are submitted to the Hive Server

Serving layer tools: Hive

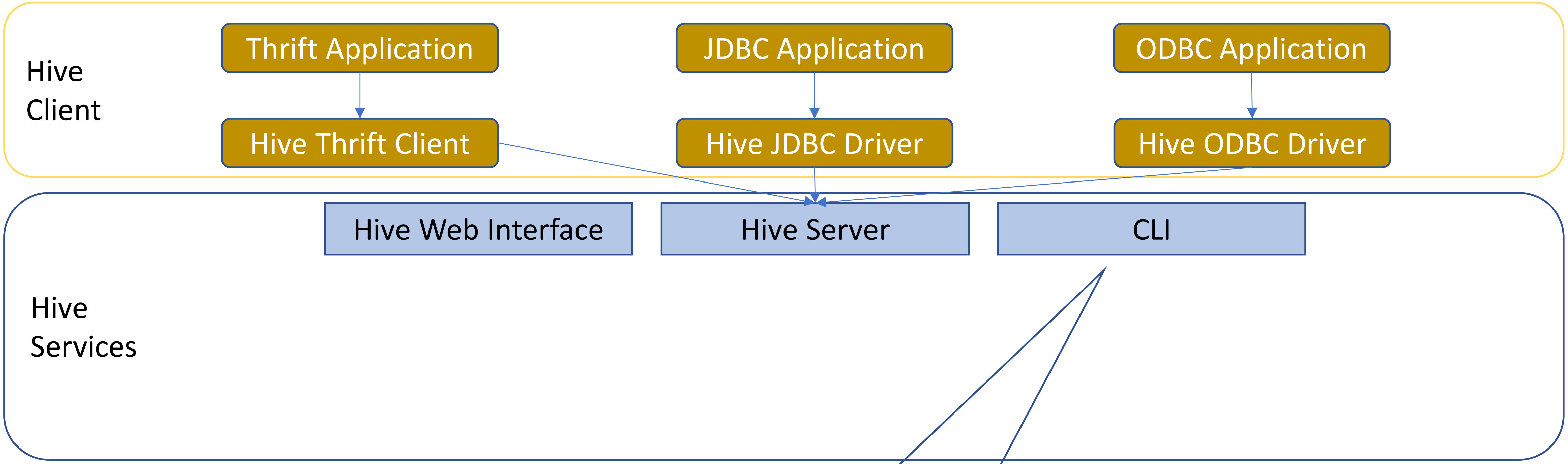
Architecture



GUI is provided to execute Hive queries

Serving layer tools: Hive

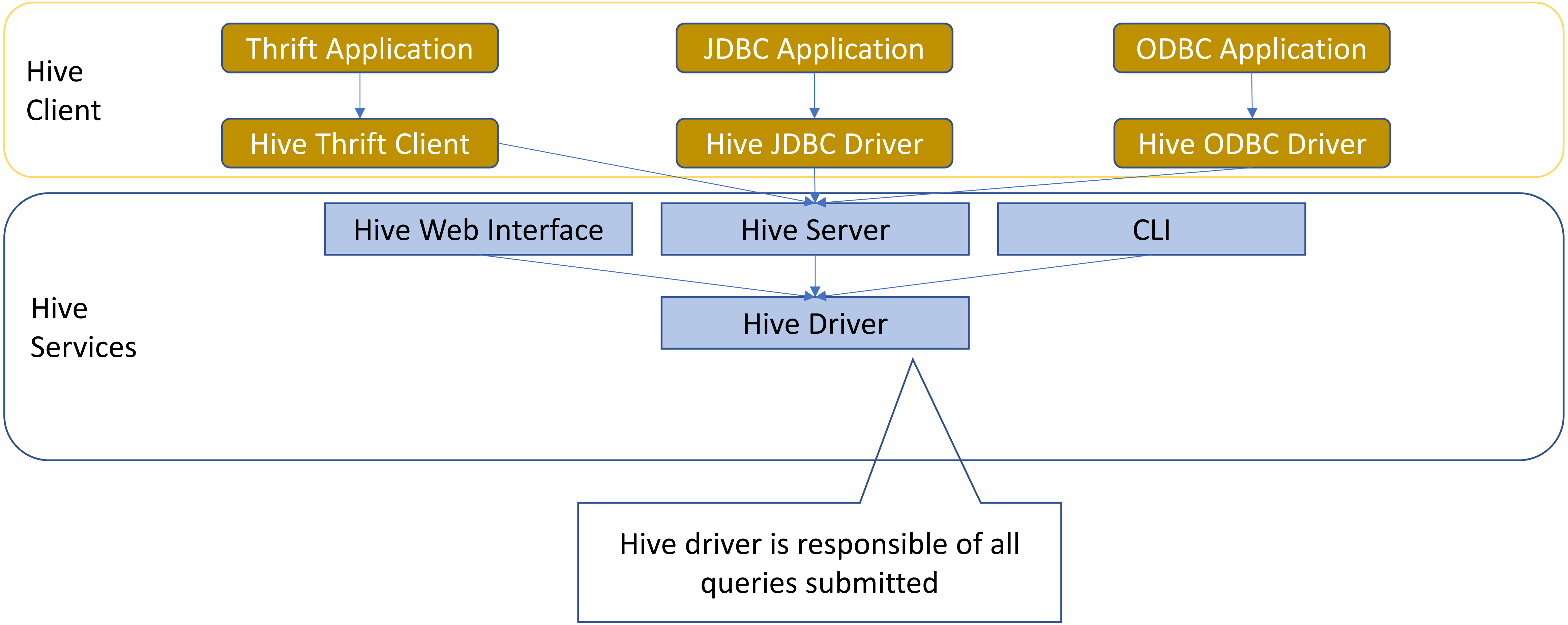
Architecture



Commands are executed directly in CLI

Serving layer tools: Hive

Architecture



Hive Client

Thrift Application

Hive Thrift Client

JDBC Application

Hive JDBC Driver

ODBC Application

Hive ODBC Driver

Hive Services

Hive Web Interface

Hive Server

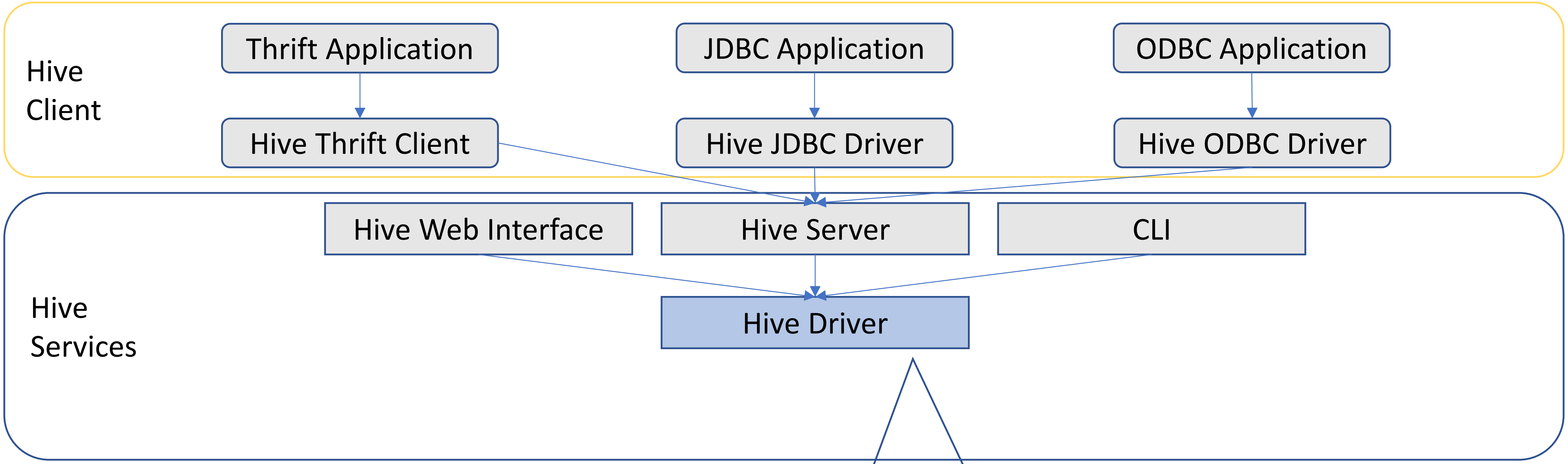
CLI

Hive Driver

Hive driver is responsible of all queries submitted

Serving layer tools: Hive

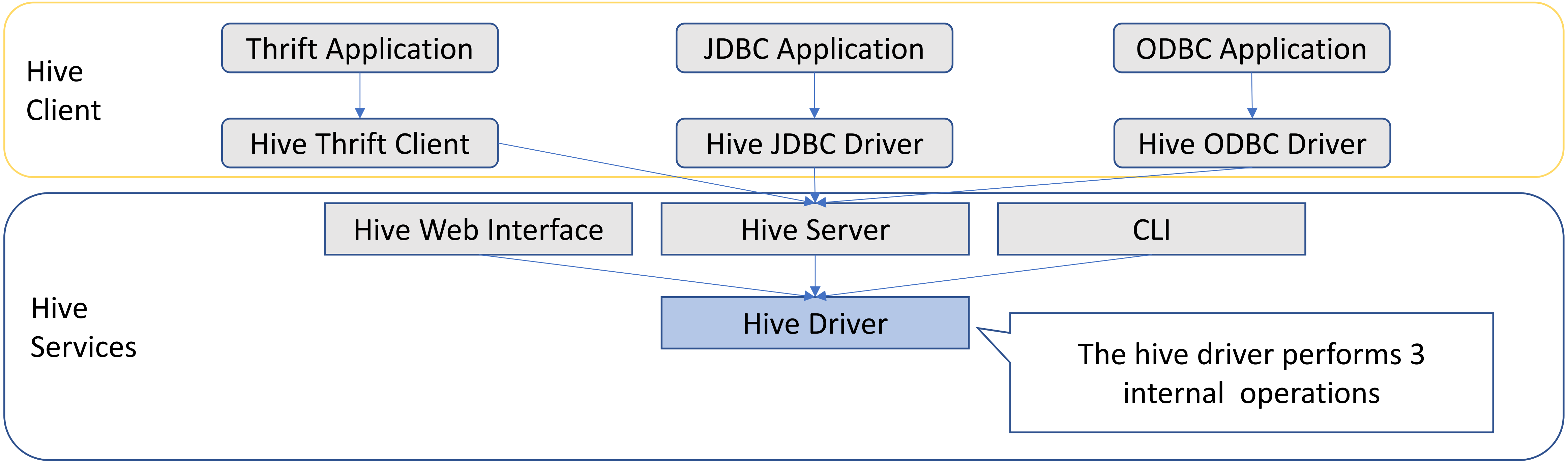
Architecture



The hive driver performs 3 internal operations

Serving layer tools: Hive

Architecture



Hive Client

Thrift Application

Hive Thrift Client

JDBC Application

Hive JDBC Driver

ODBC Application

Hive ODBC Driver

Hive Services

Hive Web Interface

Hive Server

CLI

Hive Driver

The hive driver performs 3 internal operations

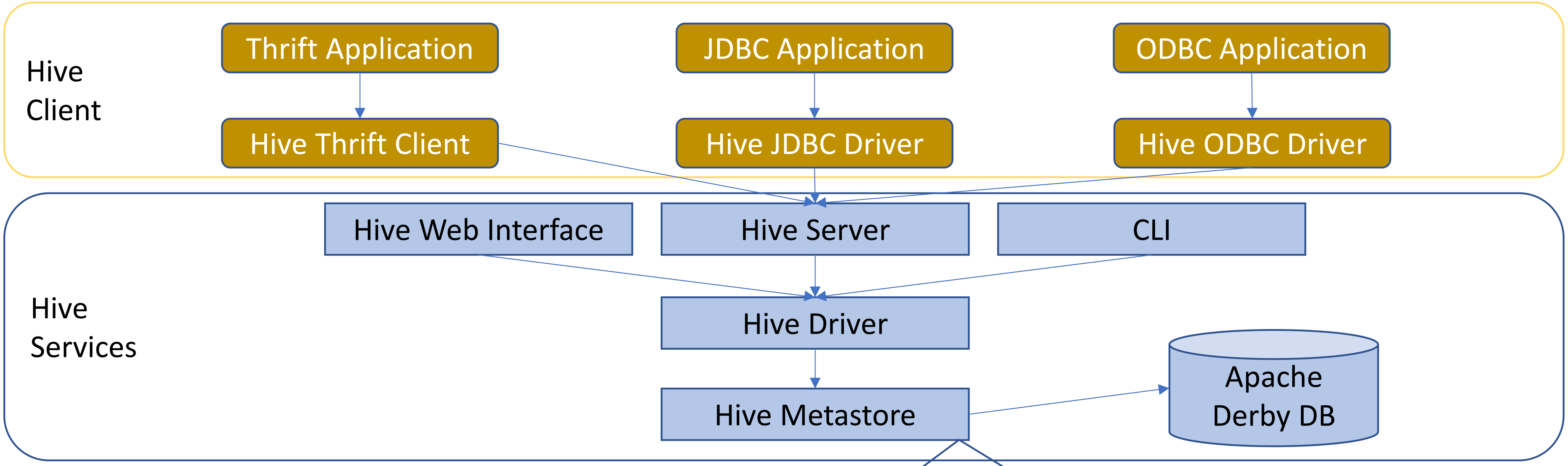
1. Compiler
Hive driver passes the query to the compiler where it is checked and analyzed

2. Optimizer
An optimized logical plan in the form of MapReduce and HDFS tasks is generated

3. Executor
Tasks are executed

Serving layer tools: Hive

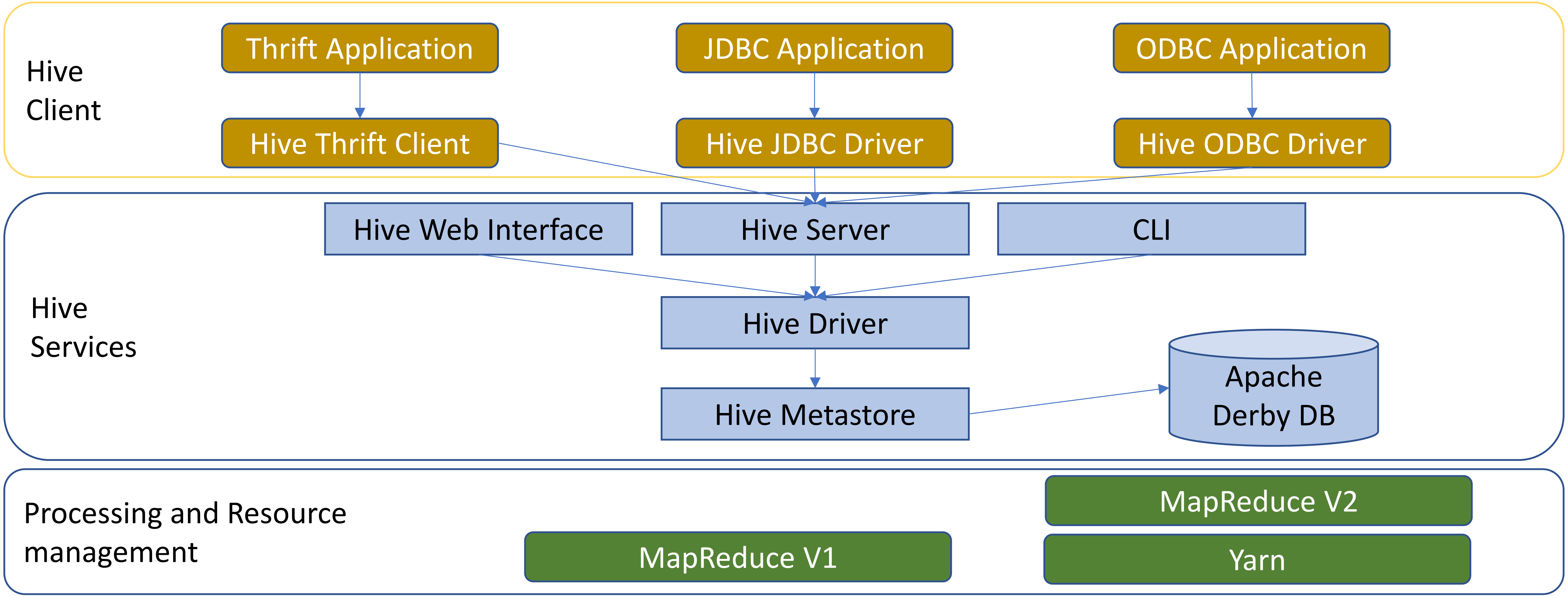
Architecture



The Metastore is a repository for Hive metadata. It stores data for hive tables. Metadata can be table metadata (name, location, schema, etc.), partition metadata (location, properties, values), database metadata (name, location, properties), or other (permissions, functions and macros, etc.).

Serving layer tools: Hive

Architecture



Hive Client

Thrift Application

JDBC Application

ODBC Application

Hive Thrift Client

Hive JDBC Driver

Hive ODBC Driver

Hive Services

Hive Web Interface

Hive Server

CLI

Hive Driver

Hive Metastore

Apache Derby DB

Processing and Resource management

MapReduce V1

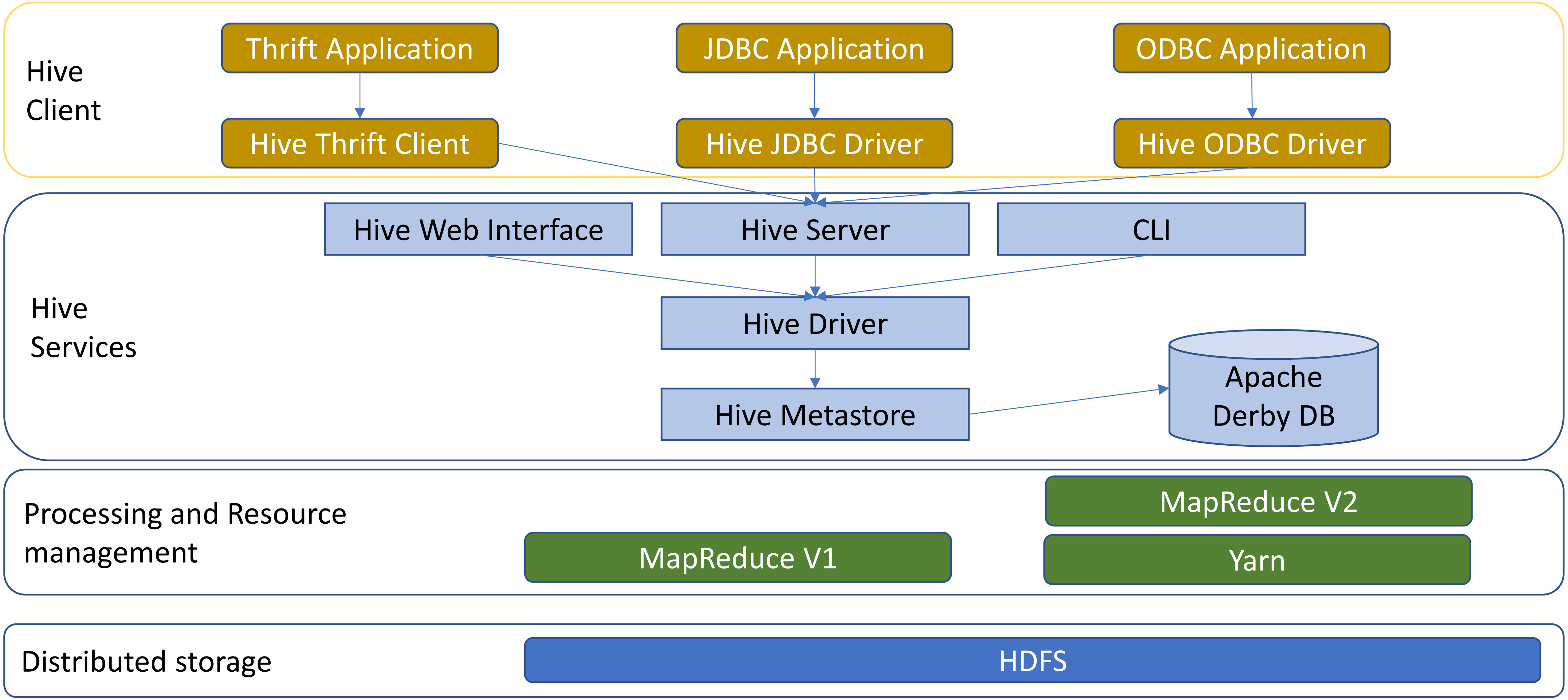
MapReduce V2

Yarn

Hive uses MapReduce to process queries

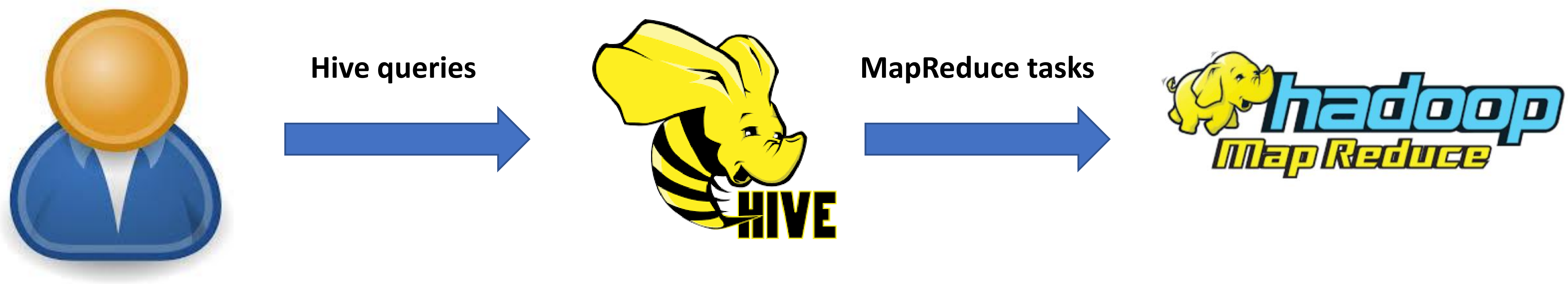
Serving layer tools: Hive

Architecture



Serving layer tools: Hive

How it works?



Hive queries

MapReduce tasks

Serving layer tools: Hive

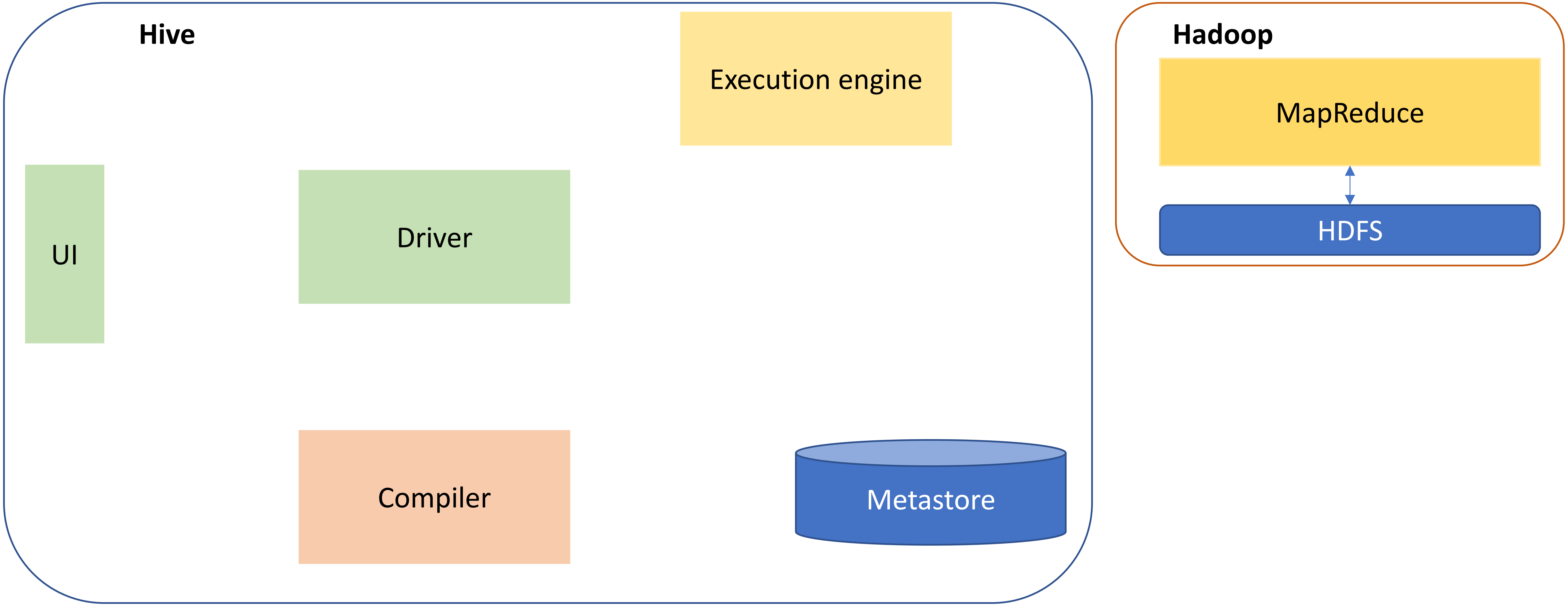
How it works?

Hive

Hadoop

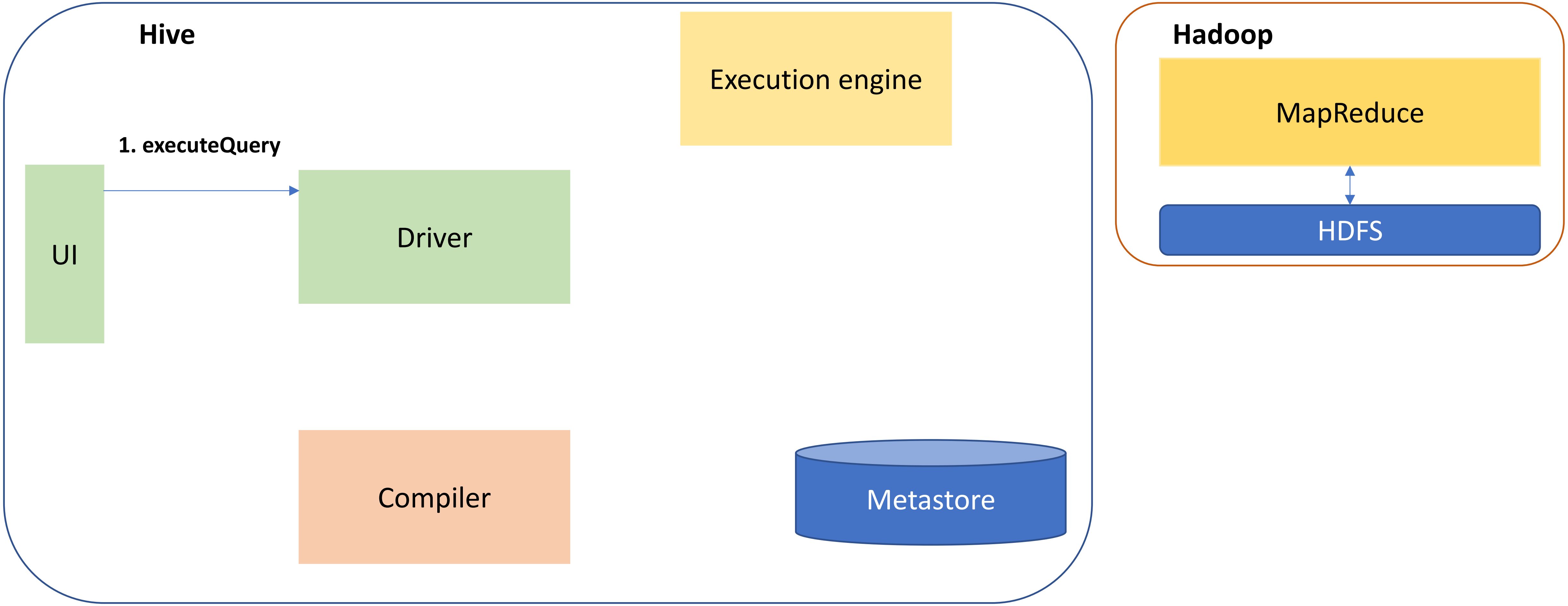
Serving layer tools: Hive

How it works?



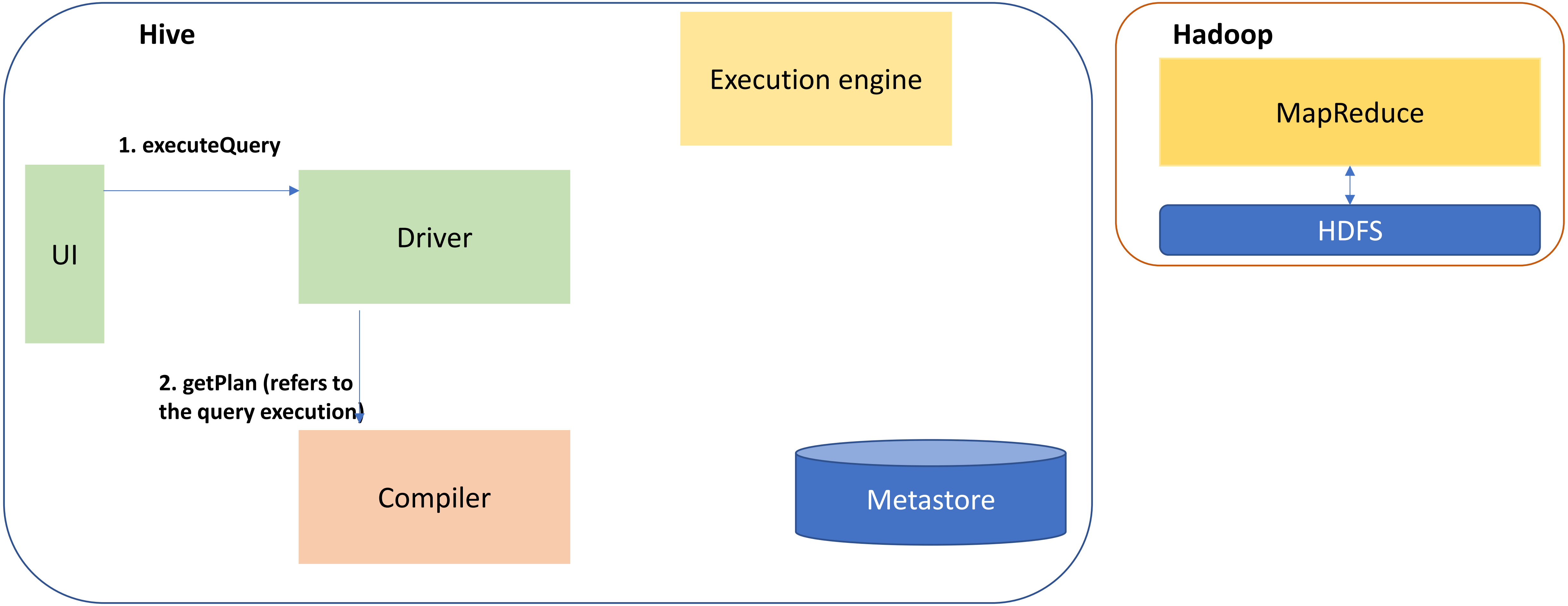
Serving layer tools: Hive

How it works?



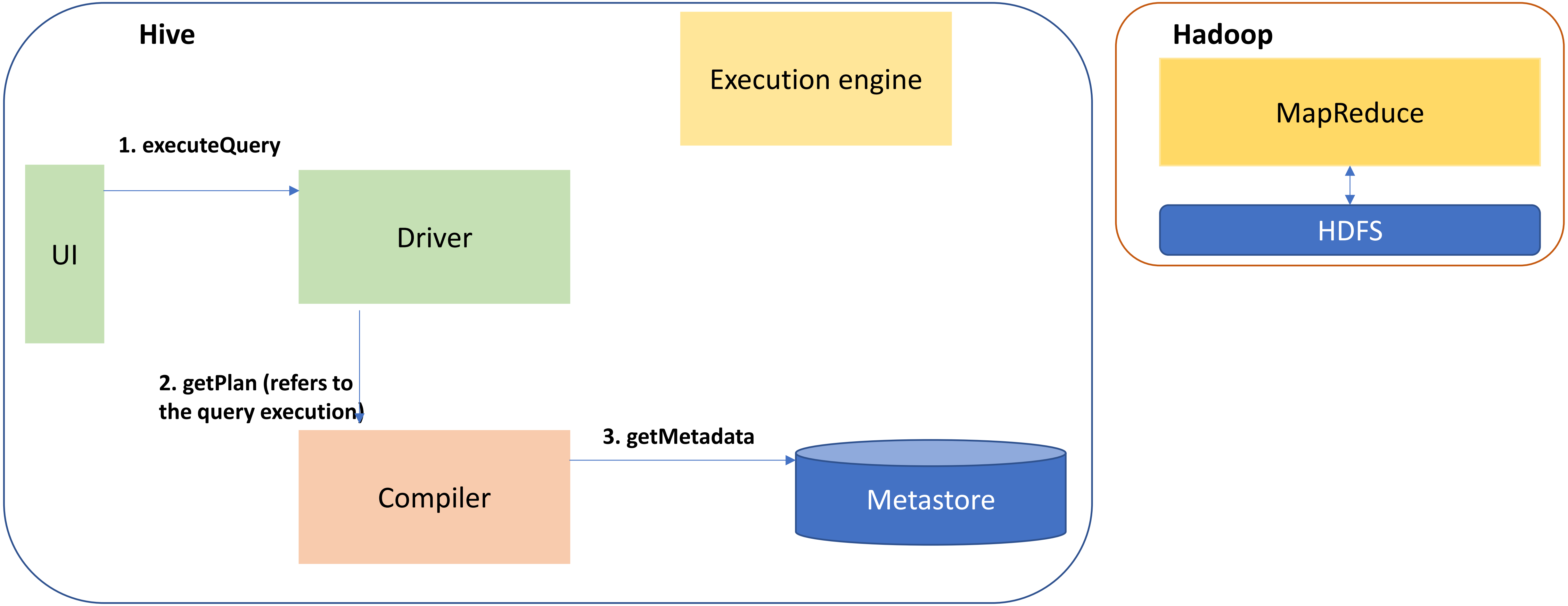
Serving layer tools: Hive

How it works?



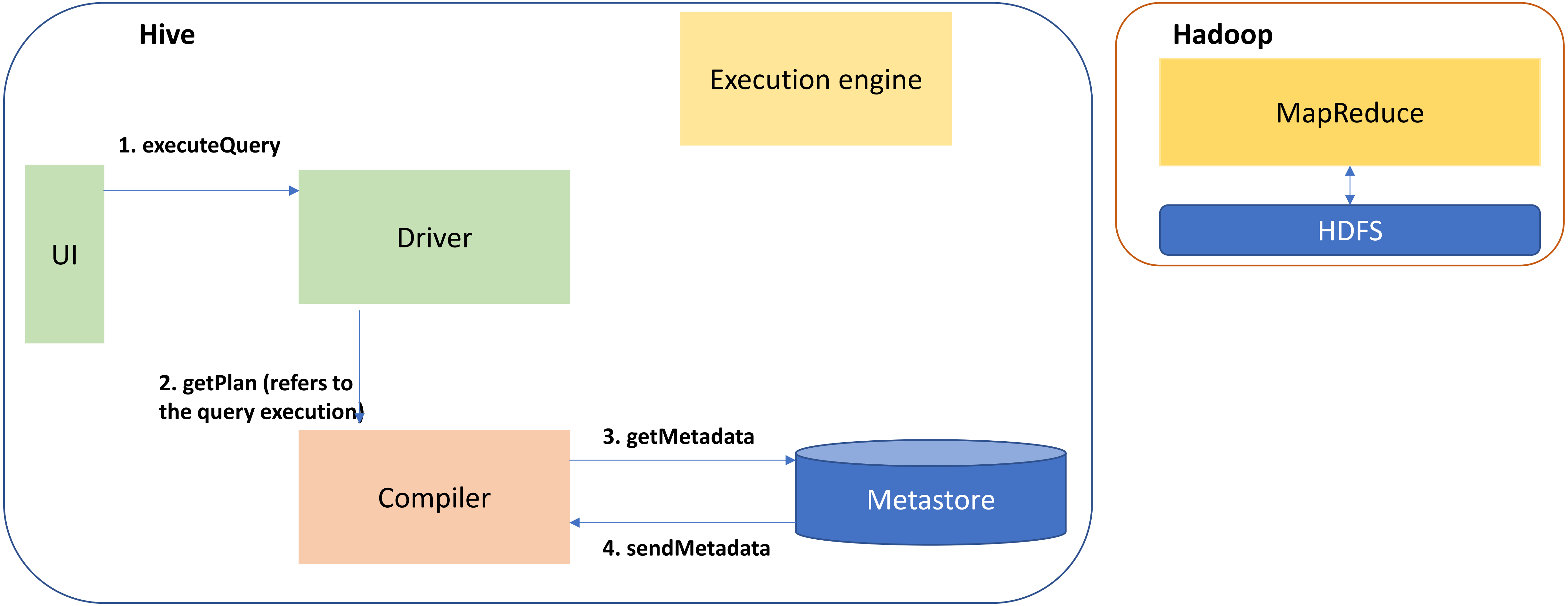
Serving layer tools: Hive

How it works?



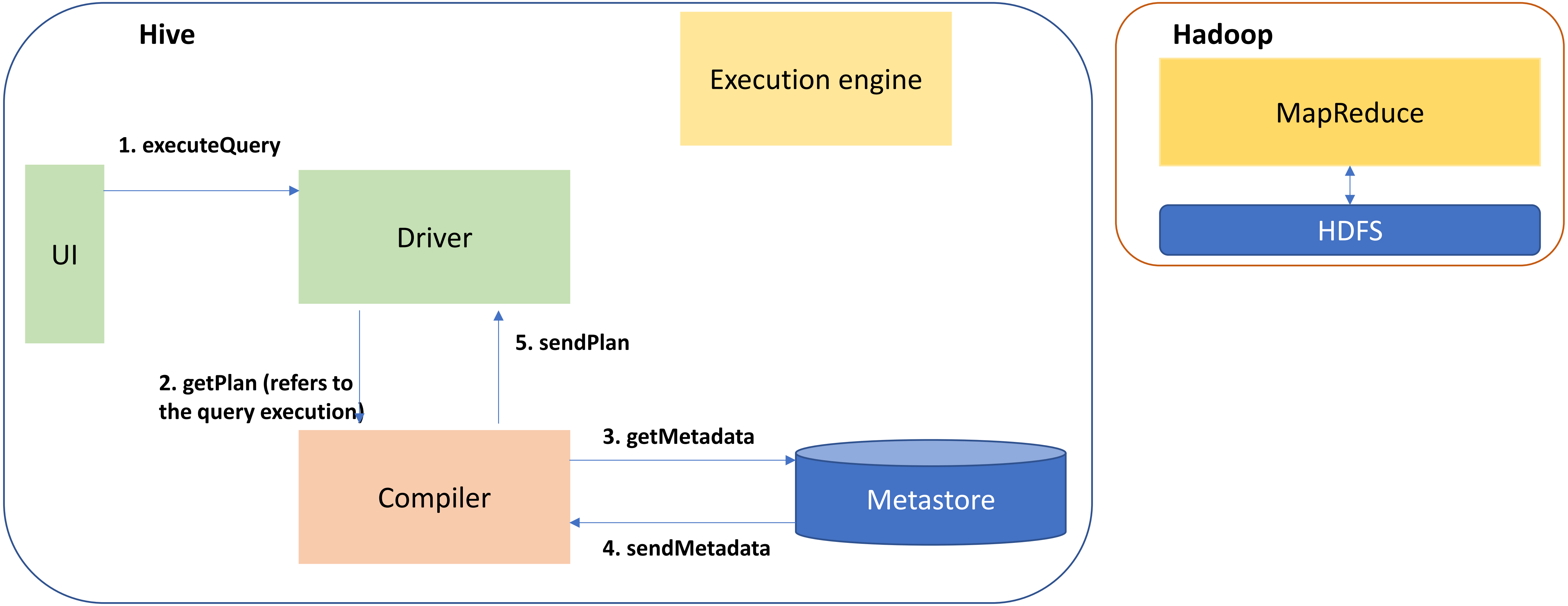
Serving layer tools: Hive

How it works?



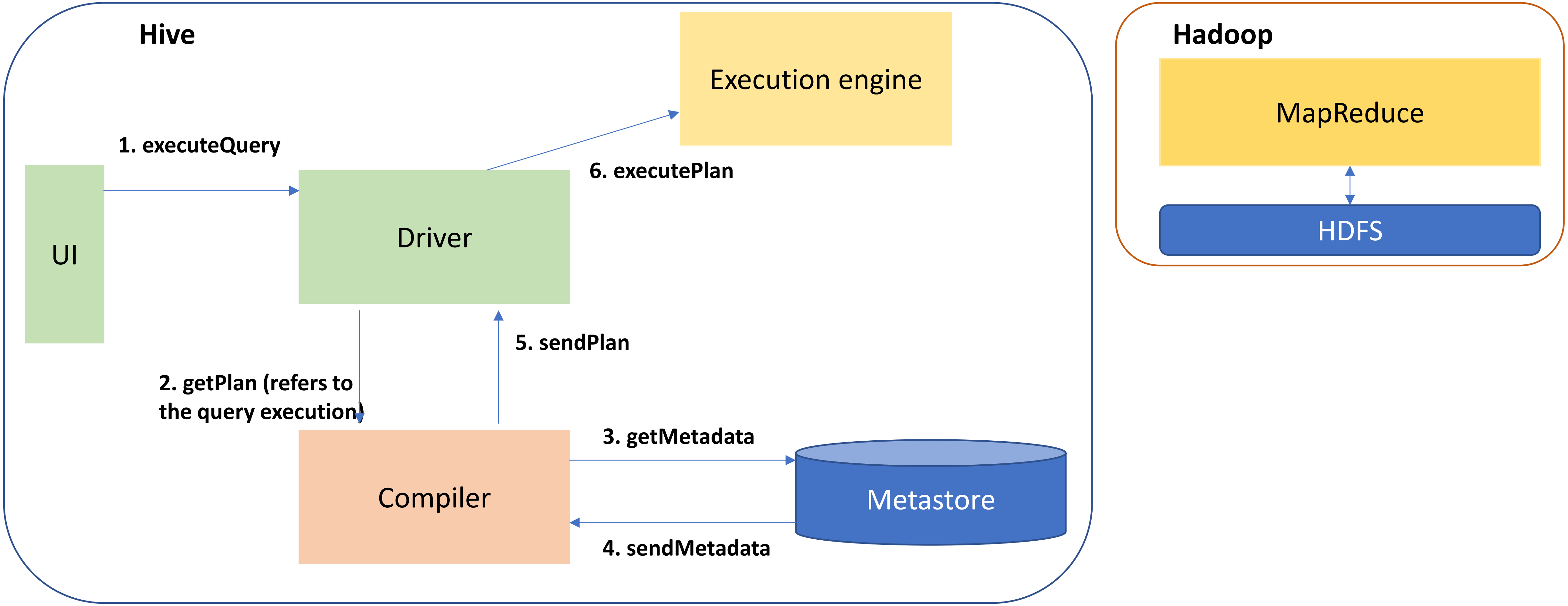
Serving layer tools: Hive

How it works?



Serving layer tools: Hive

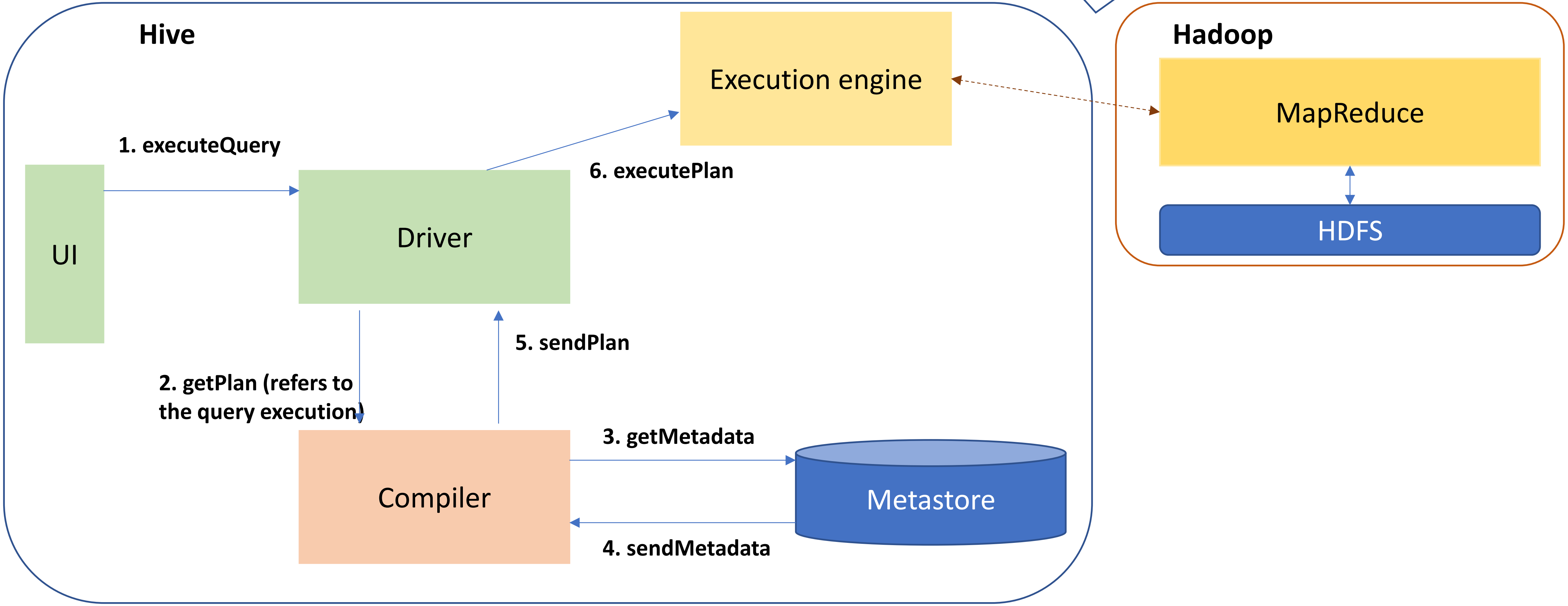
How it works?



Serving layer tools: Hive

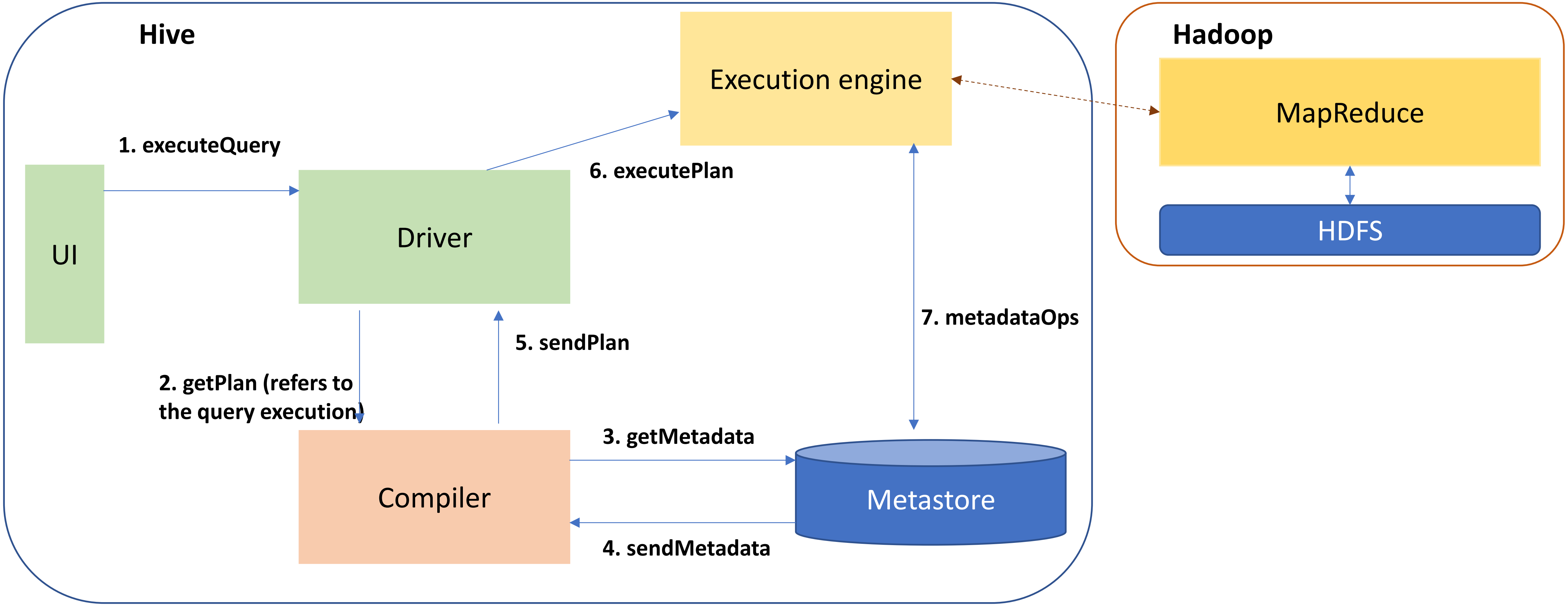
How it works?

Execution engine acts as a bridge between Hive and Hadoop to process the queries



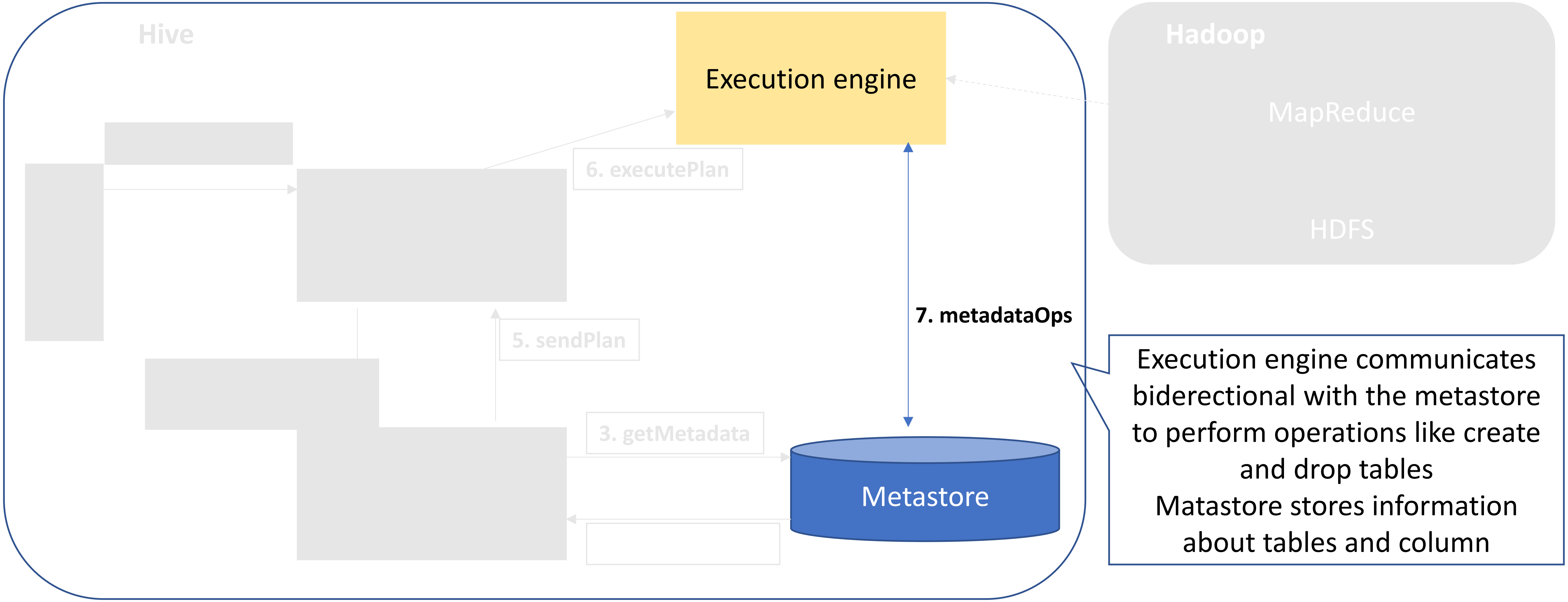
Serving layer tools: Hive

How it works?



Serving layer tools: Hive

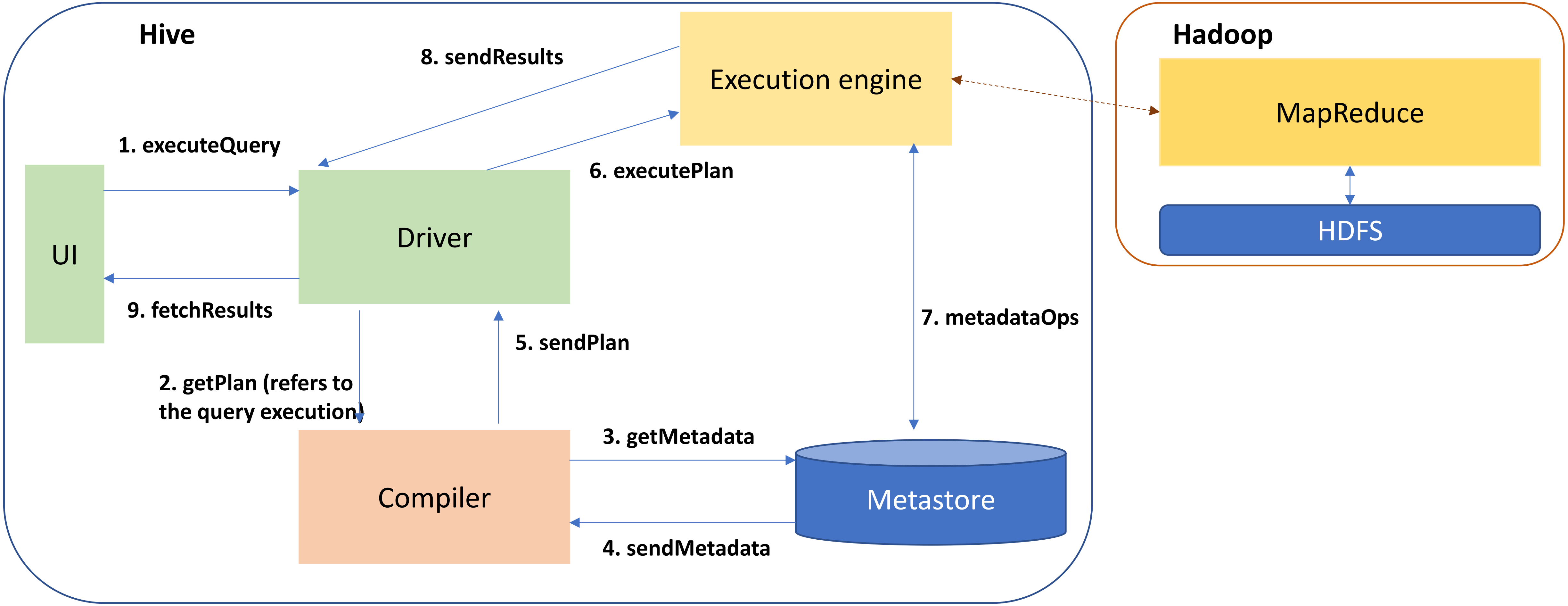
How it works?



Execution engine communicates biderectional with the metastore to perform operations like create and drop tables
Matastore stores information about tables and column

Serving layer tools: Hive

How it works?

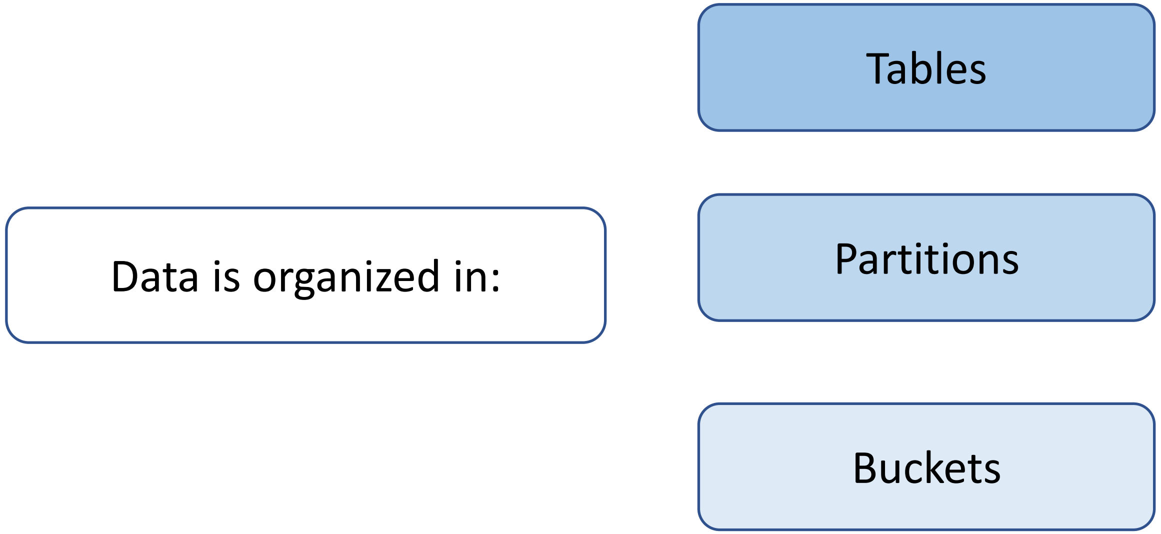


Serving layer tools: Hive

Hive data modeling

Data model: definition

A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to the properties of real-world entities.



Data is organized in:

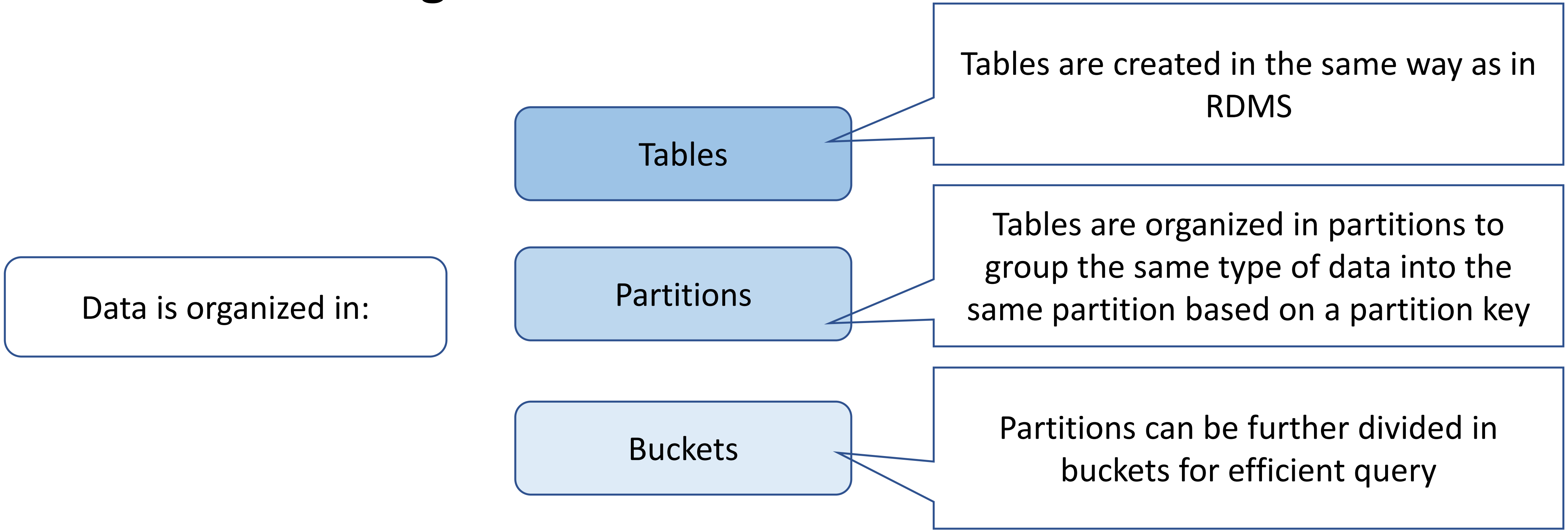
Tables

Partitions

Buckets

Serving layer tools: Hive

Hive data modeling



Data is organized in:

Tables

Tables are created in the same way as in RDMS

Partitions

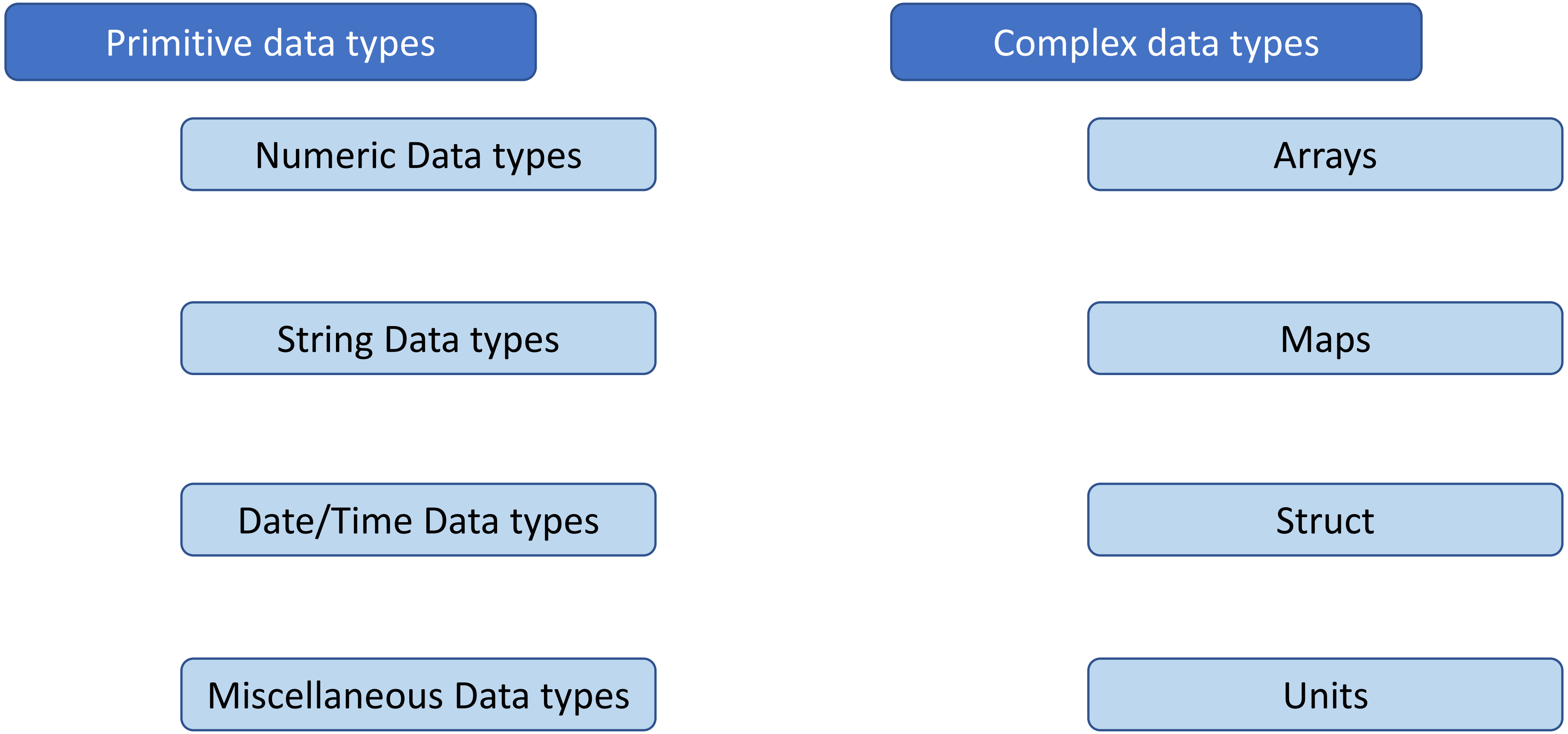
Tables are organized in partitions to group the same type of data into the same partition based on a partition key

Buckets

Partitions can be further divided in buckets for efficient query

Serving layer tools: Hive

Data types supported by Hive



Primitive data types

Numeric Data types

String Data types

Date/Time Data types

Miscellaneous Data types

Complex data types

Arrays

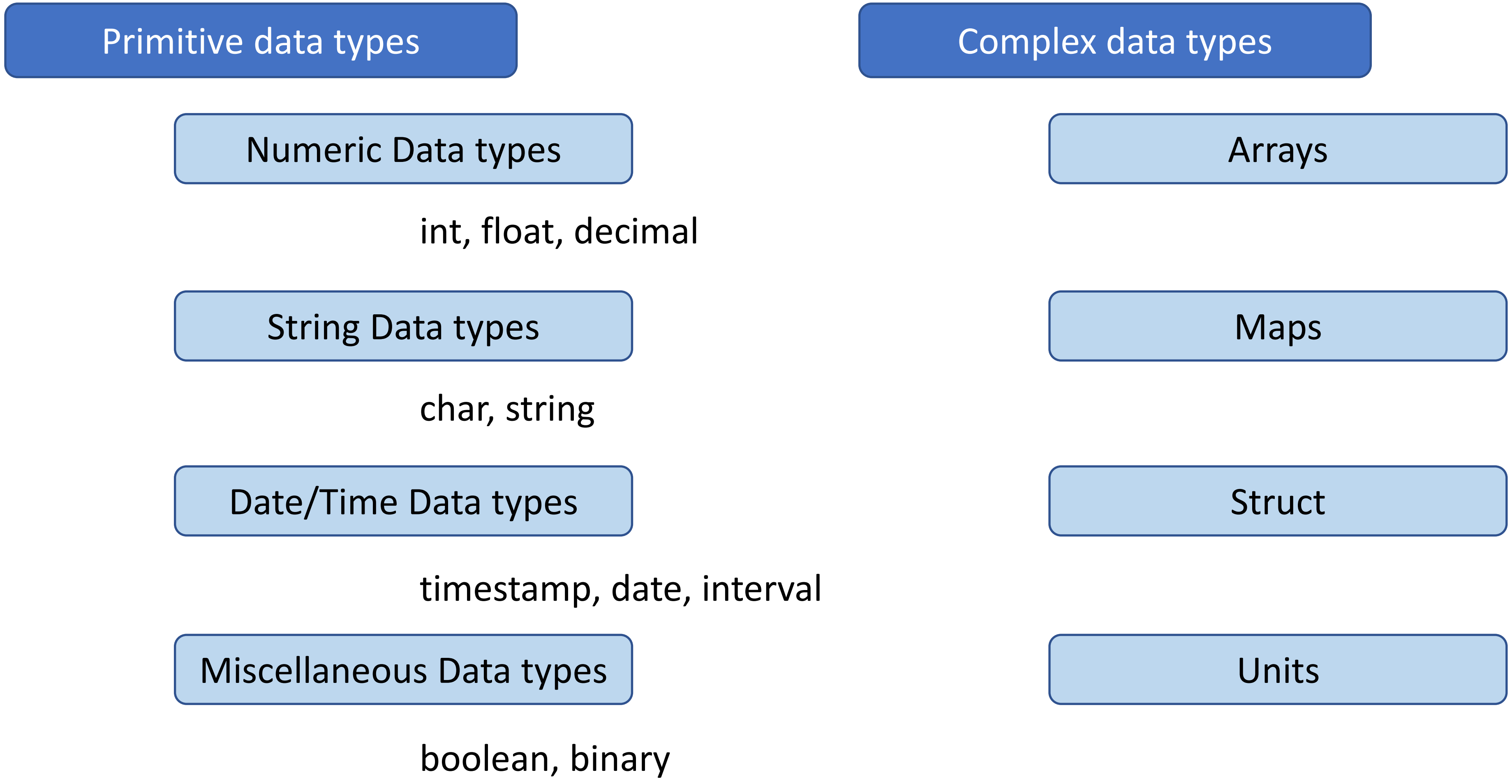
Maps

Struct

Units

Serving layer tools: Hive

Data types supported by Hive



Primitive data types

Numeric Data types

int, float, decimal

String Data types

char, string

Date/Time Data types

timestamp, date, interval

Miscellaneous Data types

boolean, binary

Complex data types

Arrays

Maps

Struct

Units

Serving layer tools: Hive

Data types supported by Hive

Primitive data types

Numeric Data types

int, float, decimal

String Data types

char, string

Date/Time Data types

timestamp, date, interval

Miscellaneous Data types

boolean, binary

Complex data types

Arrays

array <primitive data type>

Maps

Collection of key/value pairs

Struct

Collection of complex data with comment

Units

Collection of heterogeneous data types

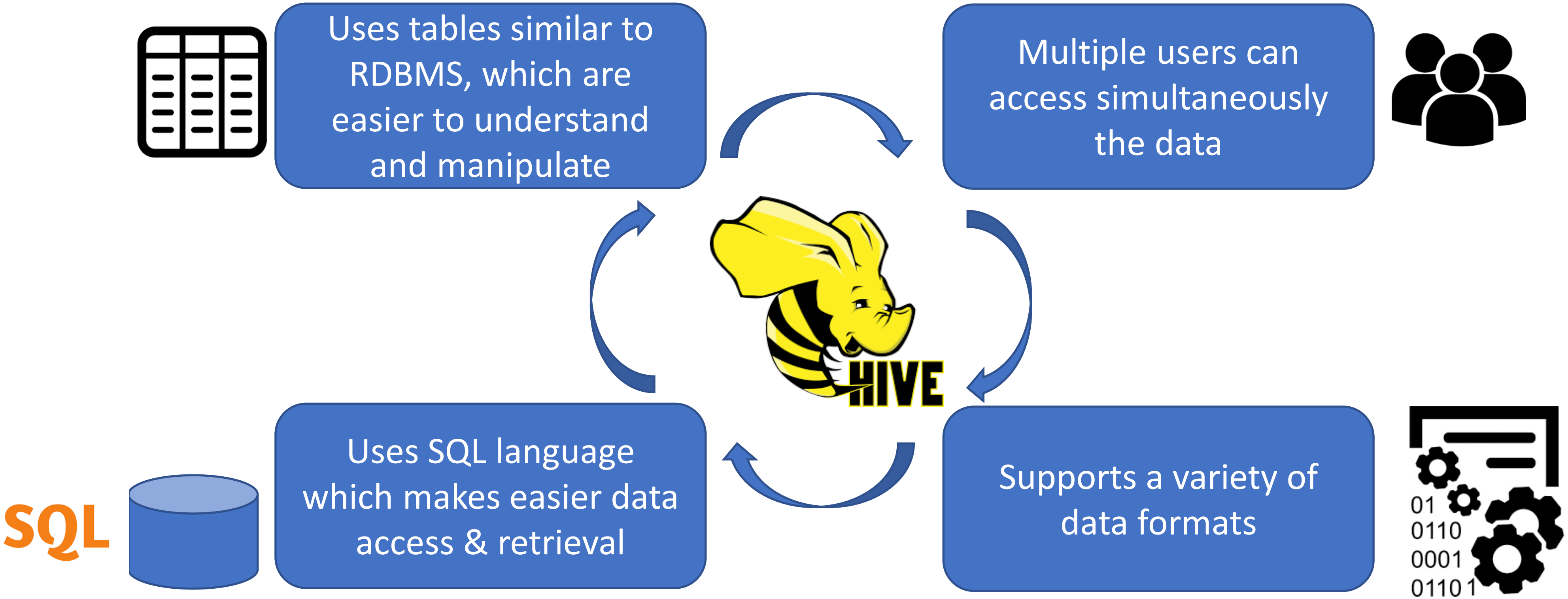
Serving layer tools: Hive

Hive vs RDMS

Hive	RDBMS
Enforces schema on read	Enforces schema on write
Size of data handled is in petabytes	Size of data handled is in terabytes
Based on the principle: **write once read many times**	Based on the principle: **read and write many times**
It is not a database, it is a data warehouse (or a query engine). **Does not store data, but enables data query**	**Stores and enable data query**
Easily scale at low costs	Does not scale easily at low costs

Serving layer tools: Hive

Key features summary



Uses tables similar to RDBMS, which are easier to understand and manipulate

Multiple users can access simultaneously the data

Supports a variety of data formats

Uses SQL language which makes easier data access & retrieval

Hive: further readings

1. Official Hive Website: https://hive.apache.org/

Serving layer – take home

- Performance metrics for the serving layer databases
- Requirements for serving layer databases
- Hive – implementation of the serving layer on top of Hadoop

Next course

- Course 7 – Speed Layer