THE DATA SCIENCE LAB
Data Wrangling with Hadoop

COM 490 – Module 2c

Week 6

Agenda 2025 - Module 2c

19.02	Introduction to Data Science with Python
26.02	(Bigger) Data Science with Python
05.03	Introduction to Big Data Technologies
12.03	Big Data Wrangling with Hadoop
19.03	Advanced Big Data Queries
26.03	Introduction to Spark
02.04	Spark Data Frames
09.04	Advanced Spark
16.04	Introduction to Stream Processing
30.04	Stream Processing with Kafka
07.05	Advanced Stream Processing
14.06	Final Project Q&A
22.05	Final Project Videos Due before midnight
28.05	Oral Sessions

Week 4 (module 2b) – Questions?

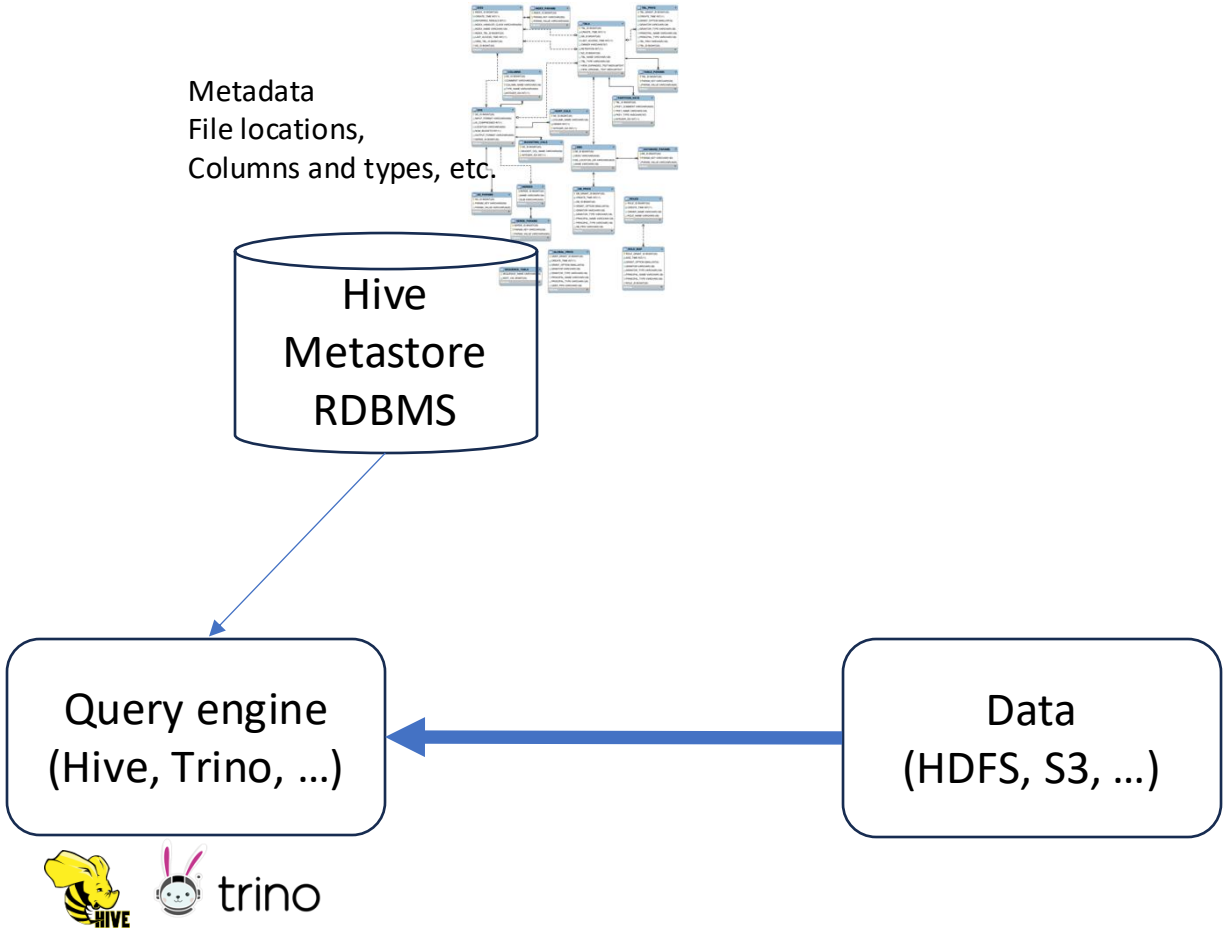
Today's Agenda

- Data table formats (Iceberg)
- NoSQL databases
- YARN – Yet Another Resource Negotiator
- Exercises
  - Processing JSON tables
  - Geospatial Functions
- Assessed projects
  - Assignment 2
  - Final project - preview

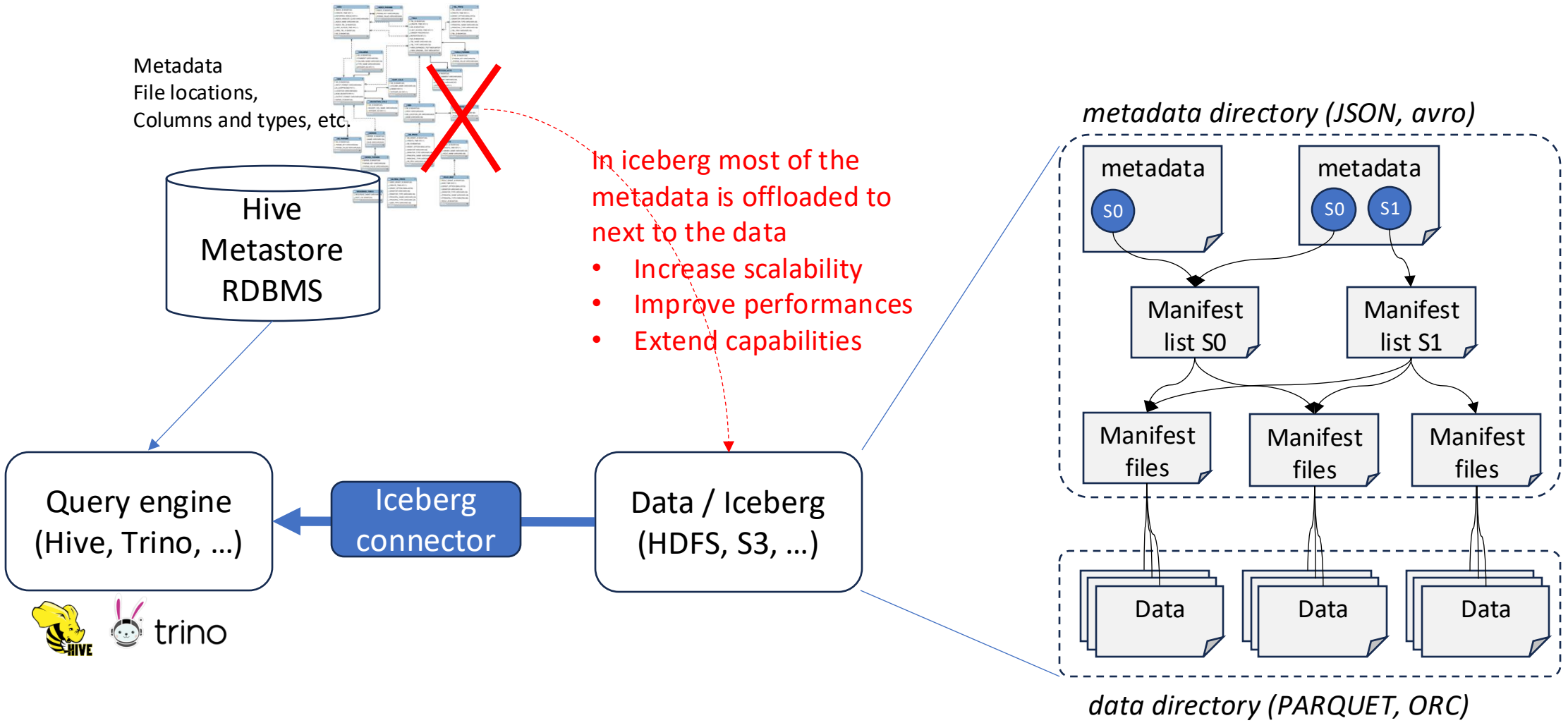
Iceberg table format

- **What is Iceberg**?
  - Open-source distributed file structure for table formats
  - Designed to manage large-scale, analytical datasets
  - Designed for and optimized for storage systems like HDFS and cloud S3
  - Provides high performance, scalability, and reliability for big data processing
  - Developed by Netflix to address **limitations** of the Hive Metastore

Hive Tables


Metadata
File locations,
Columns and types, etc.
Hive
Metastore
RDBMS
Query engine
(Hive, Trino, …)
Data
(HDFS, S3, …)
HIVE
trino

Iceberg Tables



Iceberg Tables

- **Key Features**
  - **Optimized for Big Data**: Seamlessly integrates with Spark, Hive, Trino and other big data processing engines
  - **Schema Evolution**: allows schema changes without breaking existing data or queries (including adding columns etc)
  - **Partitioning Flexibility**: Supports dynamic partitioning
    - Instead of hive partitioning hardcoded in the directory hierarchy: `year=2024/month=01`
    - Partitioning is dynamic, `year(pub_date)`, and layout is managed by iceberg (hidden)
  - **Versioned Data**: Maintains historical data versions, enabling time travel and rollback
    - https://trino.io/docs/current/connector/iceberg.html#time-travel-queries
  - **Independent Metadata Management**: Separates metadata management for better performance and scalability

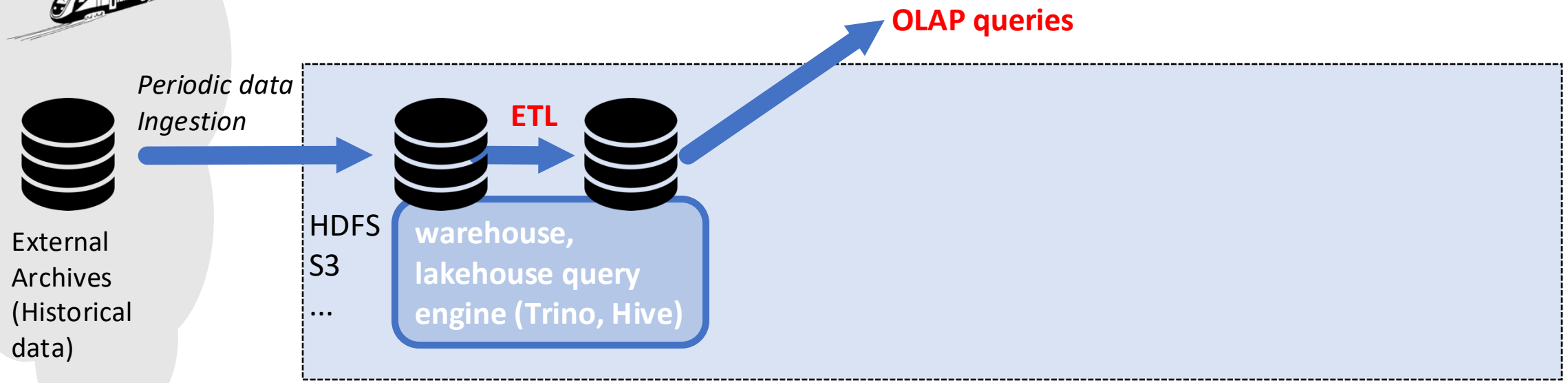
See Also

- Delta Lake: https://docs.delta.io/latest/delta-intro.html (from Databricks, 2017)
- Zarr: https://zarr.dev/
  - Designed for high dimensional scientific data (NetCDF, HDF5), e.g. used with xarray

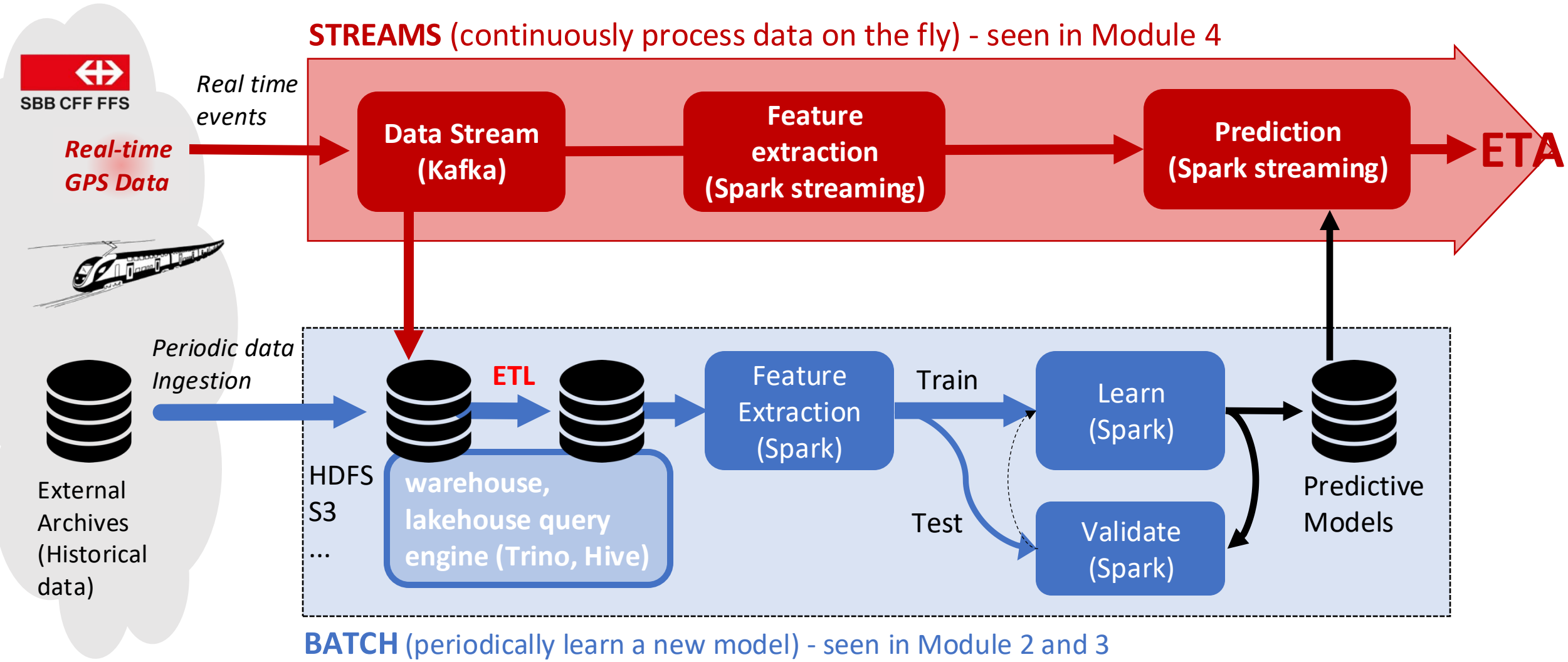
Where are we in our big data journey?

- **Ingestion** (HDFS, S3) - typically periodic (weekly, monthly)
- **Storage** - data warehouse/lakehouse (schema on read)
- **Transformation (ETL)**
  - Convert to efficient format (e.g. CSV/JSON to PARQUET/ORC)
  - Types conversions (strings to integers/timestamps), cleansing, aggregations, …
- **OLAP queries** – ad-hoc aggregations, analysis

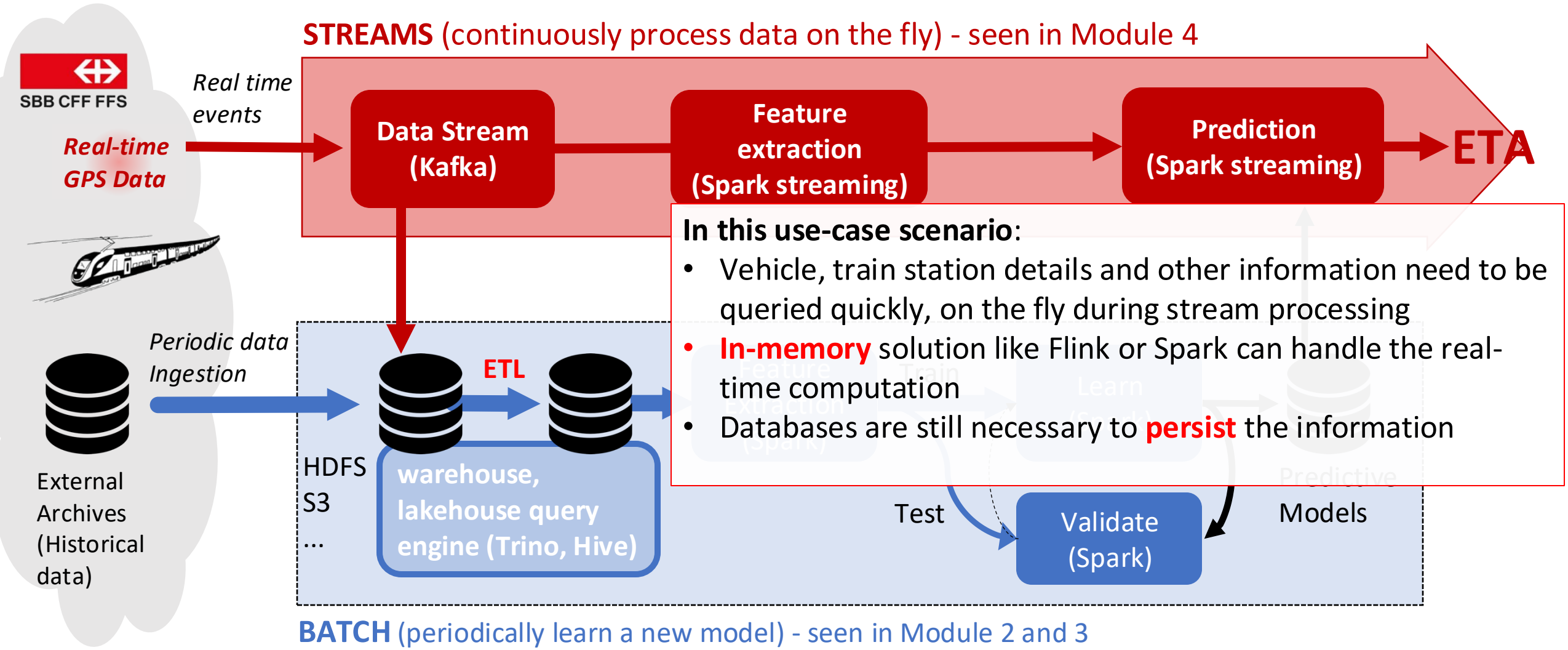
→ Scalable and efficient – cost efficient, latencies seconds/hours/days



What about real-time analytics?



What about real-time analytics?

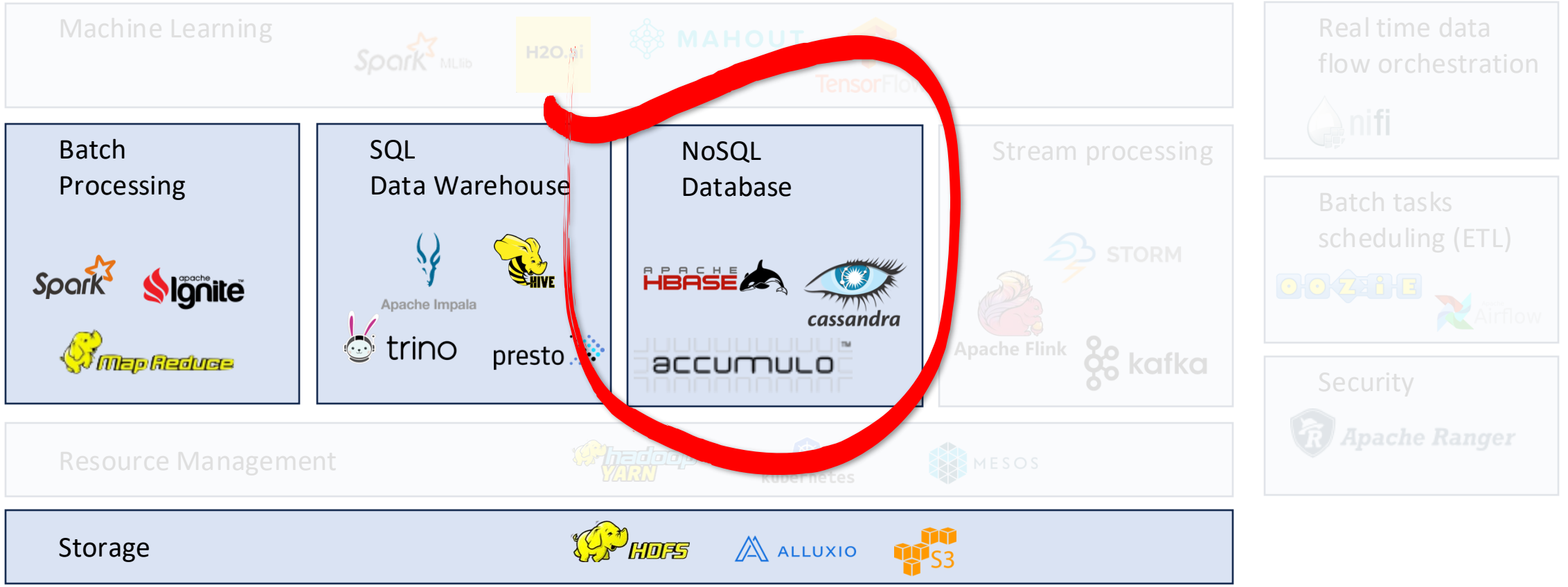


Real-Time Analytics – Beyond Data Warehouses & Lakehouses

- **Data Warehouses/Lakehouses**
  - Optimized for **OLAP queries** on large, structured datasets
  - Ideal for batch processing, if seconds/hours/days latency acceptable
  - Not well-suited for **real-time** analytics

- **Real-Time Analytics**
  - Best if data must be processed **immediately** as it arrives, often with **millisecond-level latency**
    - For real-time decision-making to provide immediate insights
  - Requires **streaming data platforms** (e.g., **Apache Kafka**, **Apache Flink**, **AWS Kinesis**)
  - Requires database designed to retrieve persisted information quickly and efficiently!

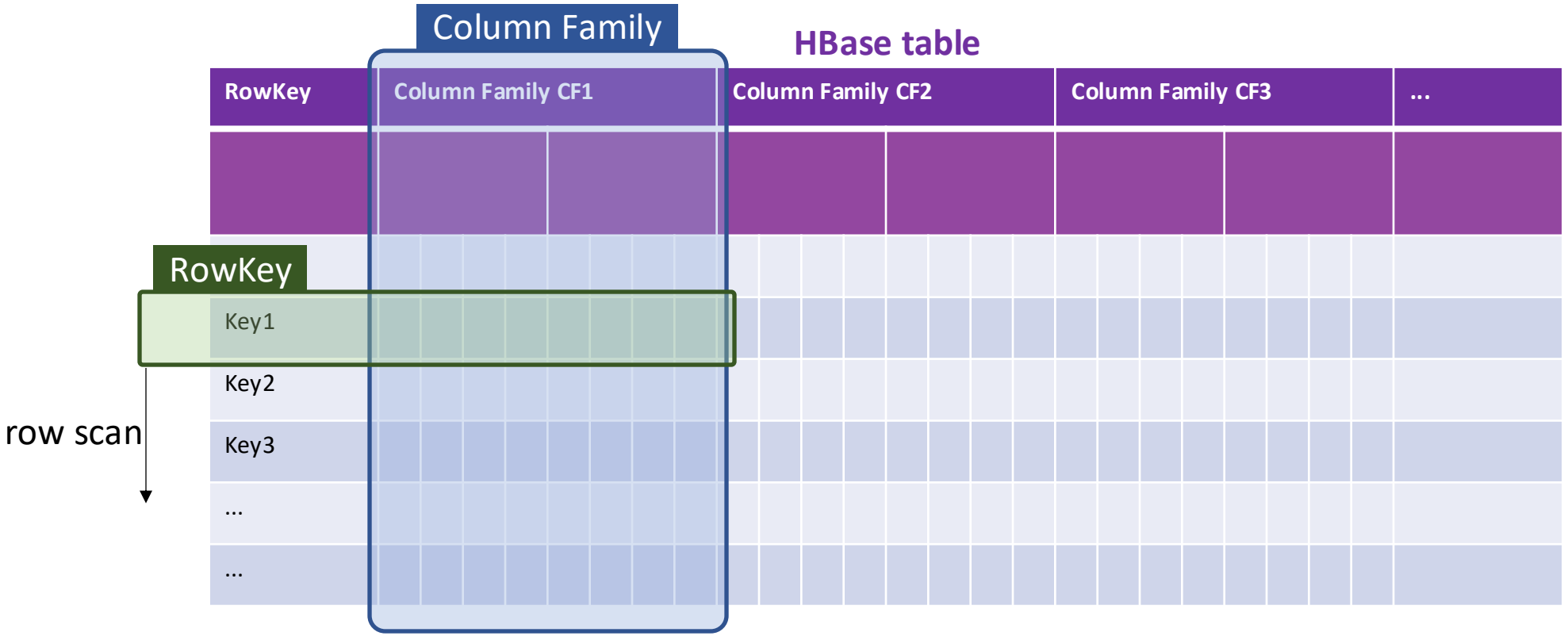
Addressing the Big Data Challenge – NoSQL



NoSQL - HBase

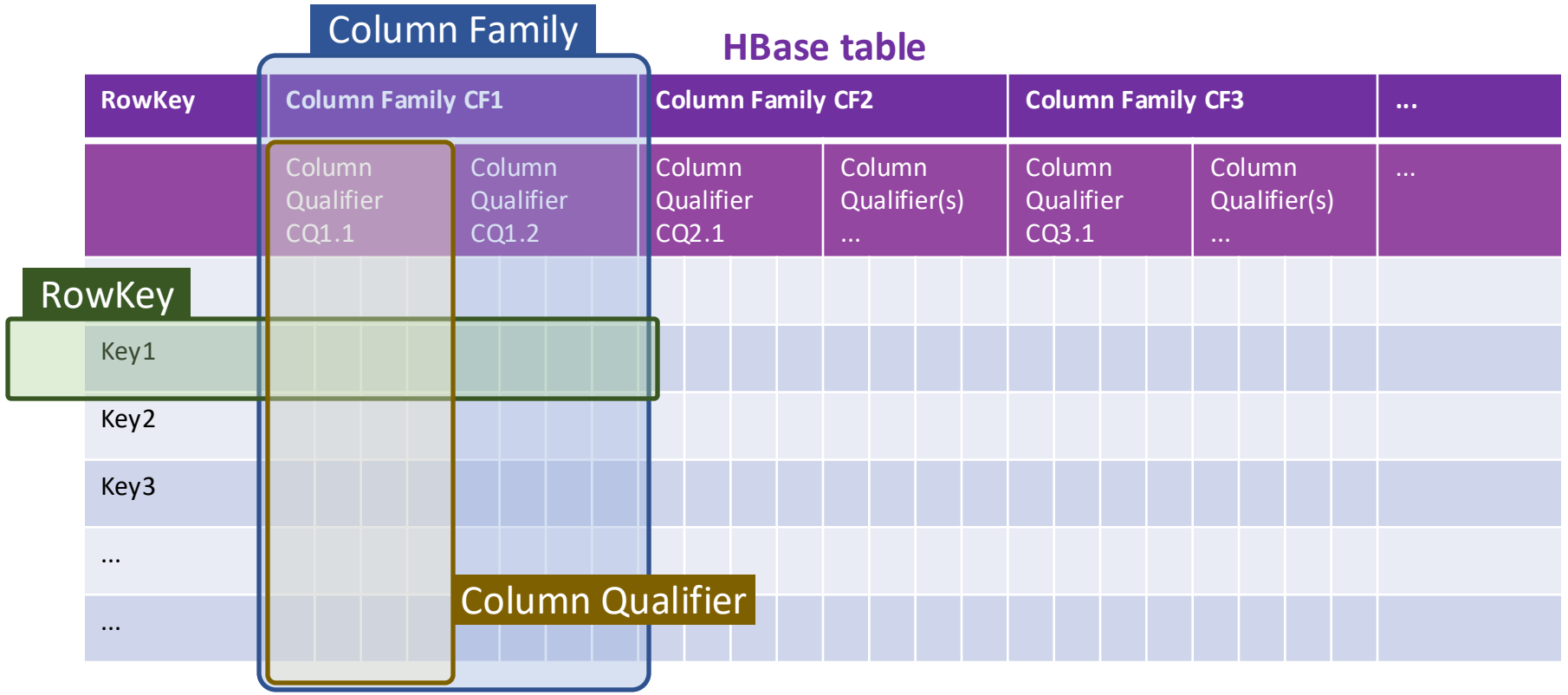
- **Key-Value Store**: Functions like a distributed dictionary
- **Indexed by RowKey & Column Families**: Column families are defined at table creation and represent a physical storage unit in the table
- **Dynamic Columns**: Columns (names and values) are created on the fly when rows are updated or created. Each column family can contain many columns
- **Sparse**: Empty columns do not consume space; they are created when data is inserted
- **Wide Column Store**: Can handle billions of rows, each potentially having millions of columns
- **Versioned**: Each column can store multiple versions; changes create new versions
- **Cell Access**: Data is accessed using the tuple:
  namespace:table {row-key, column-family:column-name, version}

HBase Column Oriented Data Model



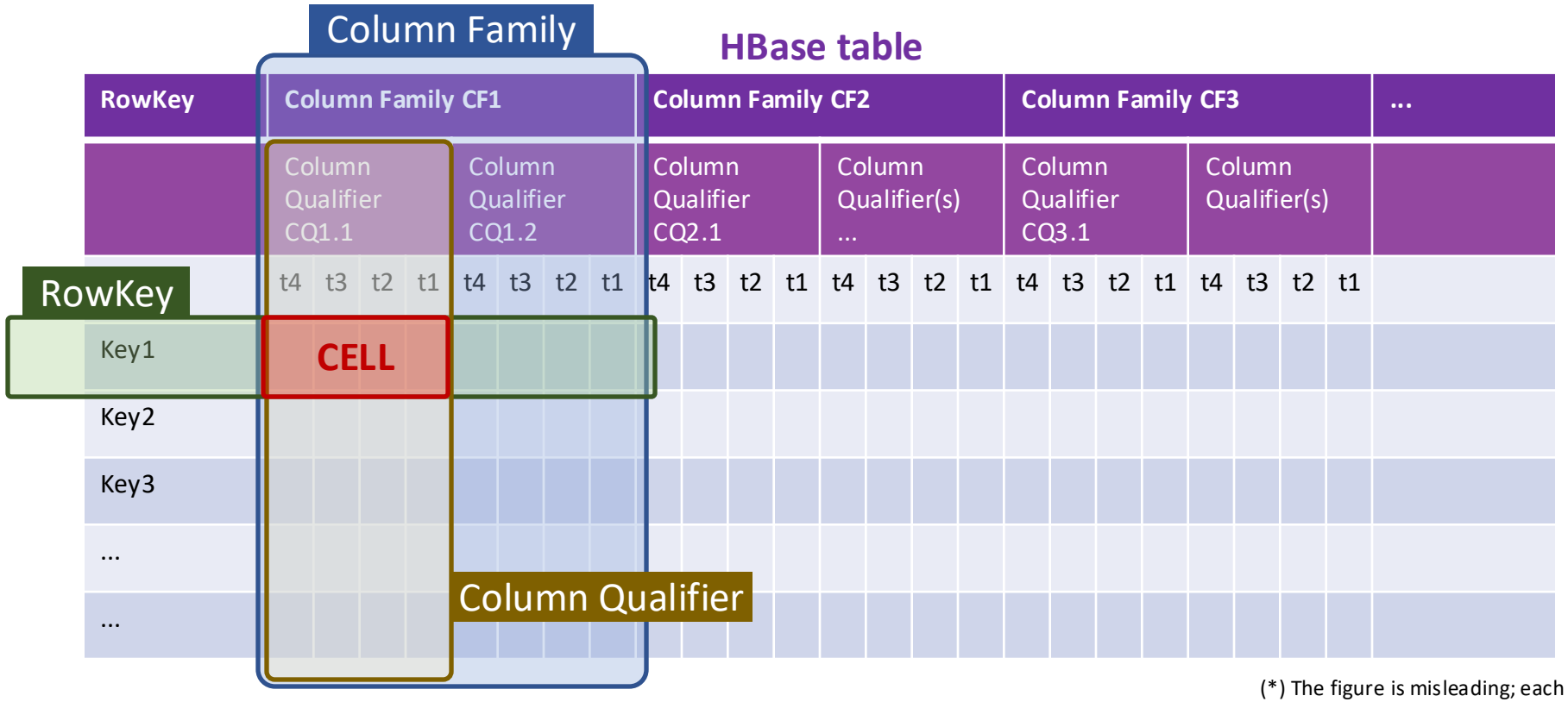
- **Column families** (CF) group related columns that should be stored together for I/O efficiency
- **RowKeys** uniquely identify each row in CF, stored in order to optimize retrieval

HBase Column Oriented Data Model



- **Column qualifier** are conjured-up on the fly (when you insert values)
- Technically they do not use space in a RowKey/Column Family until they are declared
- Everywhere else, they don't exist, and take no space

HBase Column Oriented Data Model



(*) The figure is misleading; each column qualifier can vary depending on the RowKey and column family

- **RowKey** + **Column Family**/**Column Qualifier** = **CELL coordinates**
- Cells store the values, along with timestamps (date and time)
- It is possible to retrieve earlier versions (earlier timestamps) of a cell's value

HBase Architecture – Regions

HBase table

RowKey	Column Family CF1								Column Family CF2								Column Family CF3								...
	Column Qualifier CQ1.1				Column Qualifier CQ1.2				Column Qualifier CQ2.1				Column Qualifier(s) ...				Column Qualifier CQ3.1				Column Qualifier(s) ...				
Region	t4	t3	t2	t1	t4	t3	t2	t1	t4	t3	t2	t1	t4	t3	t2	t1	t4	t3	t2	t1	t4	t3	t2	t1	
Key1																									
Key2																									
Key3																									
...																									
...																									

- HBase tables are divided into **regions** stored on HDFS, typically with a size of 250MB per block.
- Each **region** corresponds to a single column family and covers a specific range of sorted **RowKeys**.
- Multiple regions can be accessed concurrently, enabling efficient distributed computing.

Hbase - Summary

- Hbase Row

Row-Key	Column family cf1	Column family cf2
0123456abcdef	col1={ 1, 2, 3 }, col2={“a”, ”b” }	col3={ “v1”, “v2” }
0123456ghijklm	col1={ 4, 5, 6 }	col3={ “v3”, “v4” }
….		

- Most common DDL operations
  - create table ‘namespace:tablename’, { cf1 properties }, { cf2 properties }
  - enable/disable table ‘namespace:tablename’
  - drop tables ‘namespace:tablename’
  - list ‘namespace:.*’
- Most common DML operations
  - put ‘namespace:table’, ‘row-key’, ‘cf1:col1’, value, [‘version-ts’]
  - get ‘namespace:table’, ‘row-key’, [time range, column, versions]
  - scan ‘namespace:table’, [row range, time range, filters, ...]

HBase – “Good” key design

- Column families (and their column qualifiers) are indexed by the primary key
- Consecutive keys are stored in the same HRegion and can be efficiently accessed by a “range scan”, i.e. Returns all keys in a start/end range
- It is therefore important to design the keys according to the type of queries we intend to perform on the data
- The choice of the key is made “before-hand”

HBase – “Good” key design

- For instance, to efficiently access time series of individual sensors identified by a unique ID:
  - Key SensorID.yyyymmddHHMMSS
  - E.g. ZHLBC.20240320131500

- You can then perform a range scan, e.g. all measures of sensor ZHLBC on March 20, 2024:
  - Scan ZHLBC.20240320 will return the data (column families) of all the keys that begin with the given sequence, i.e. All the measures for of ZHLBC on 2024.03.20.

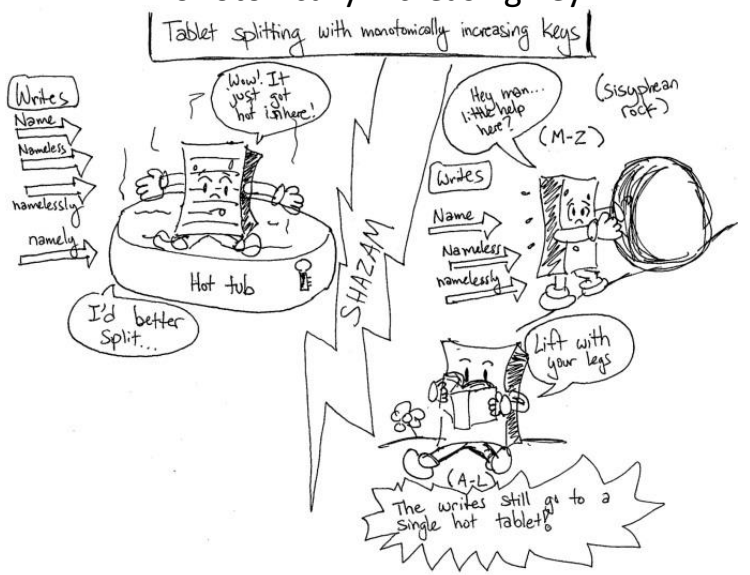
- The same key is less efficient if we want to return the measures of ALL the sensors at a particular time
  - Because the key starts with the sensor ID, we have to repeat the query for each sensors; we cannot get the data using a single range scan

HBase – Bulk Data Loading

- When loading data in bulk in HBase it is recommended to not insert monotonically increasing keys, e.g. 1,2,3,4 …
- Doing otherwise may result in resource usage “hot-spots”, i.e. Large ranges of consecutive keys will be grouped into HRegions and HRegions (i.e. HRegionServer) will be populated one at the time.

(semi) random key

Monotonically increasing key



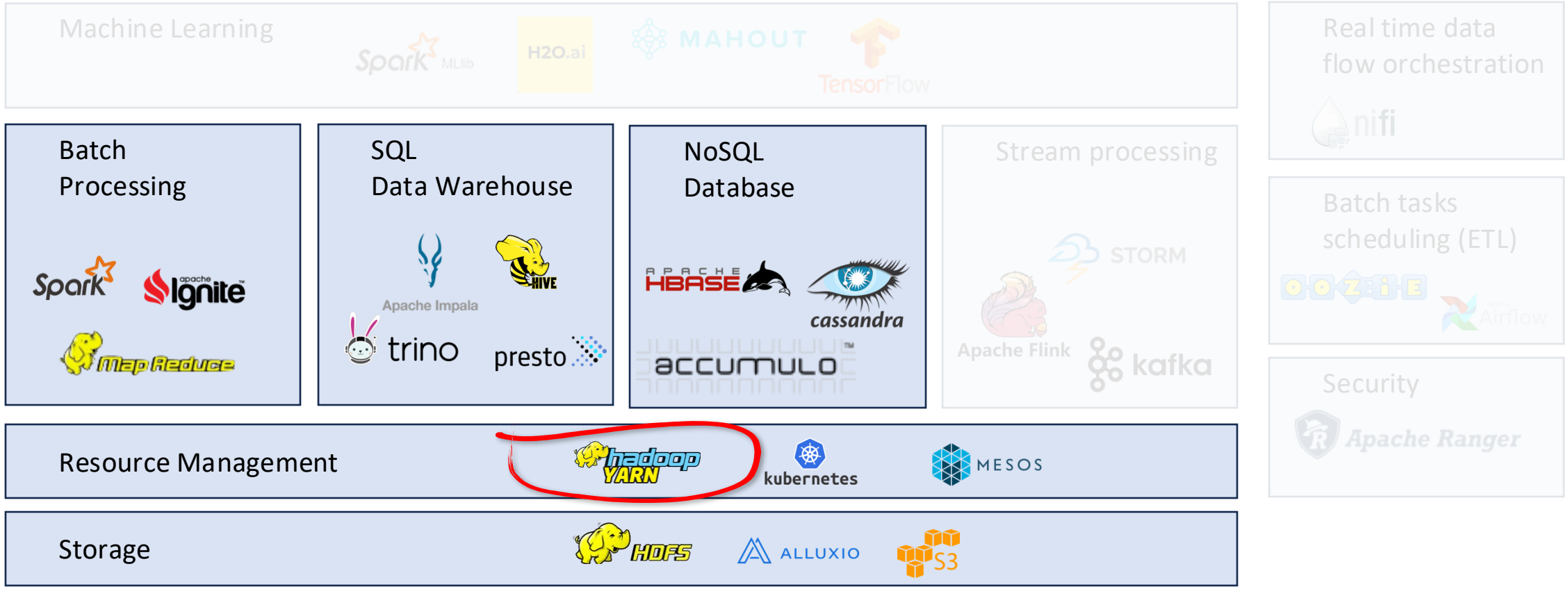
Apache HBase vs … *other* NoSQL

- Apache Cassandra
- Apache Accumulo
- MongoDB
- ScyllaDB (OSS AGPL 6.2)
- Azure Cosmos DB (cloud)
- Amazon - AWS DynamoDB (cloud)
- Google BigTable (cloud)
- … to name a few

YARN

Yet Another Resource Negotiator

Addressing the Big Data Challenge – Resource Management

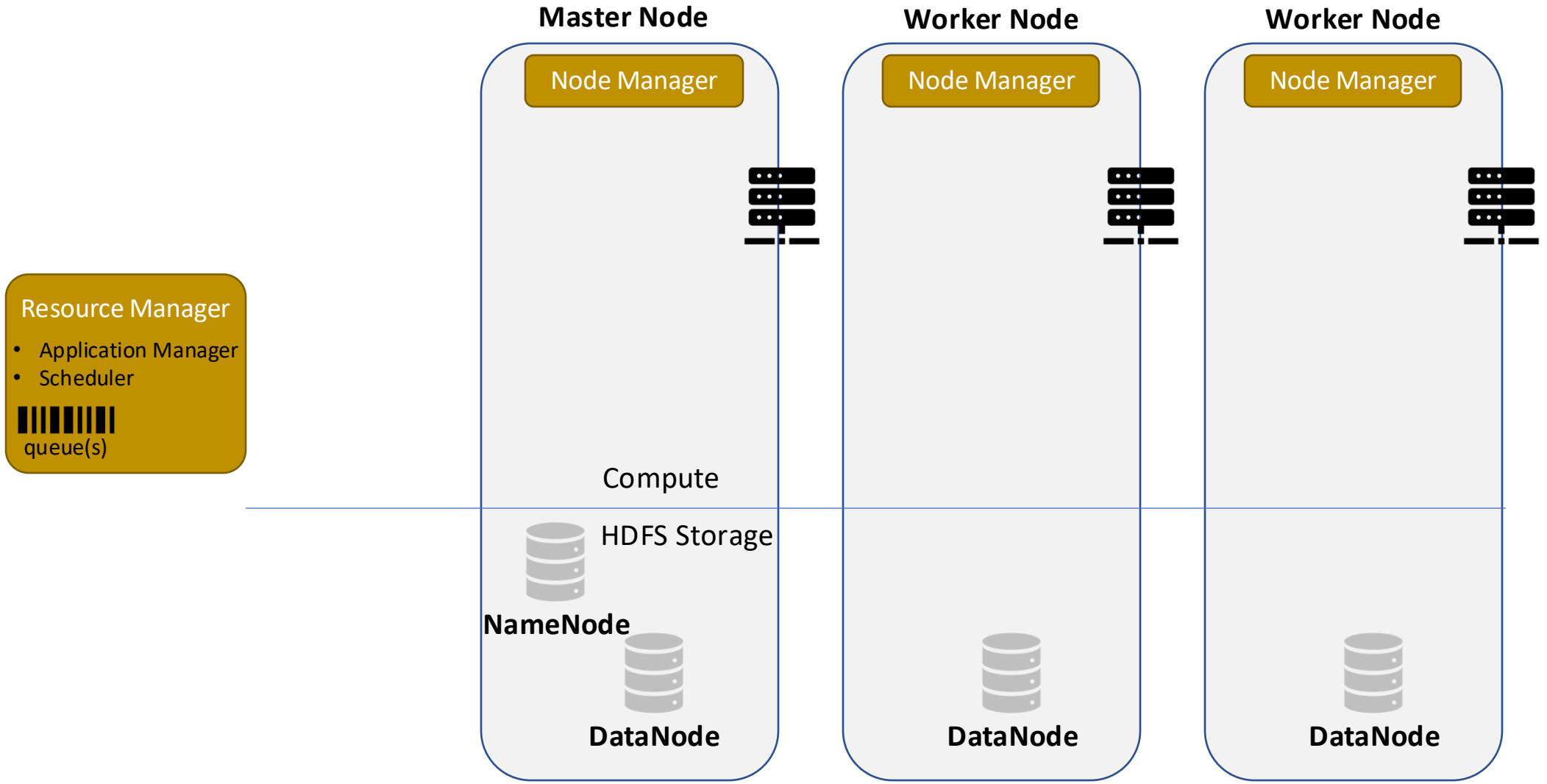


Yet Another Resource Negotiator - YARN

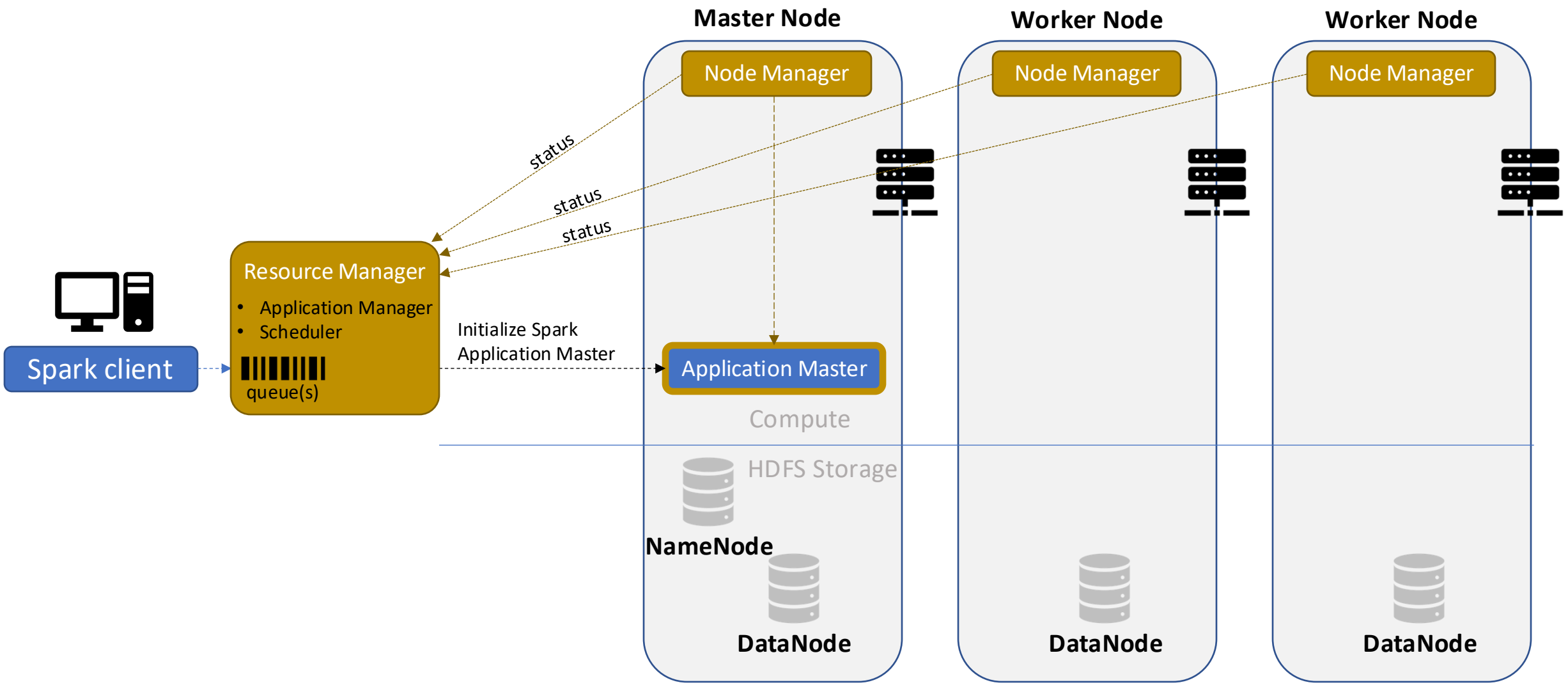
- YARN

- Role of Application Masters in YARN
  - Resource Request and Allocation
  - Job Management and Task Scheduling
  - Progress monitoring
  - Failure Recovery
  - Resource Cleanup

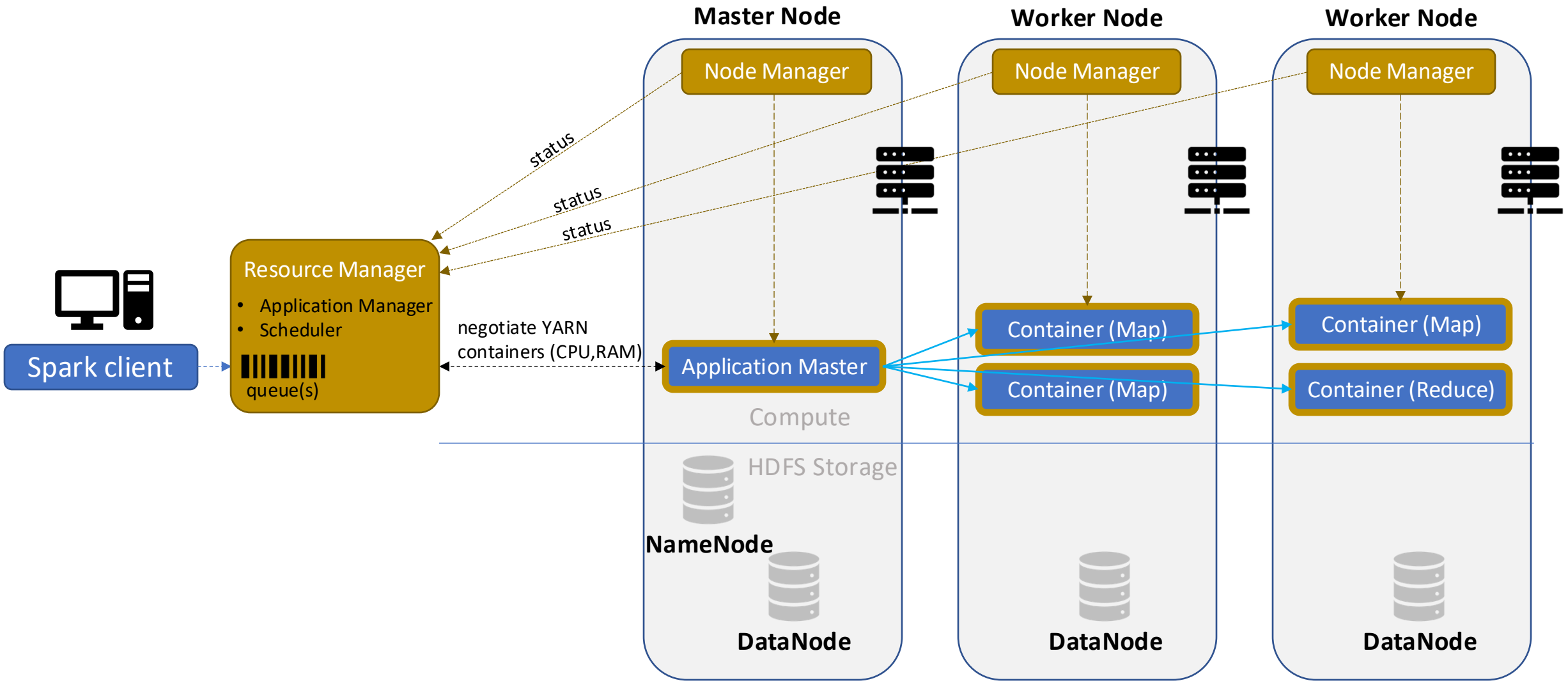
YARN


Master Node
Worker Node
Worker Node
Node Manager
Node Manager
Node Manager
Resource Manager
• Application Manager
• Scheduler
queue(s)
Compute
HDFS Storage
NameNode
DataNode
DataNode
DataNode

Spark on YARN



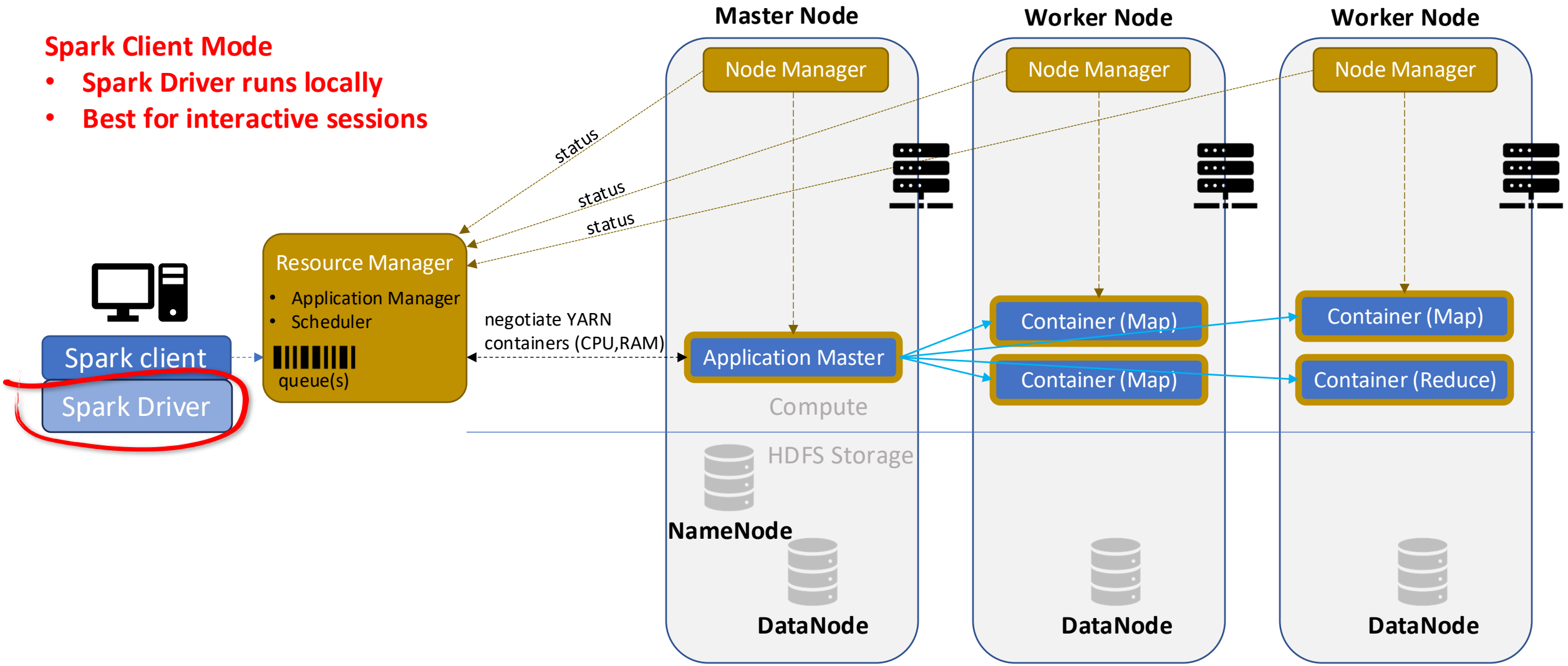
Spark on YARN



Spark on YARN

Spark Client Mode

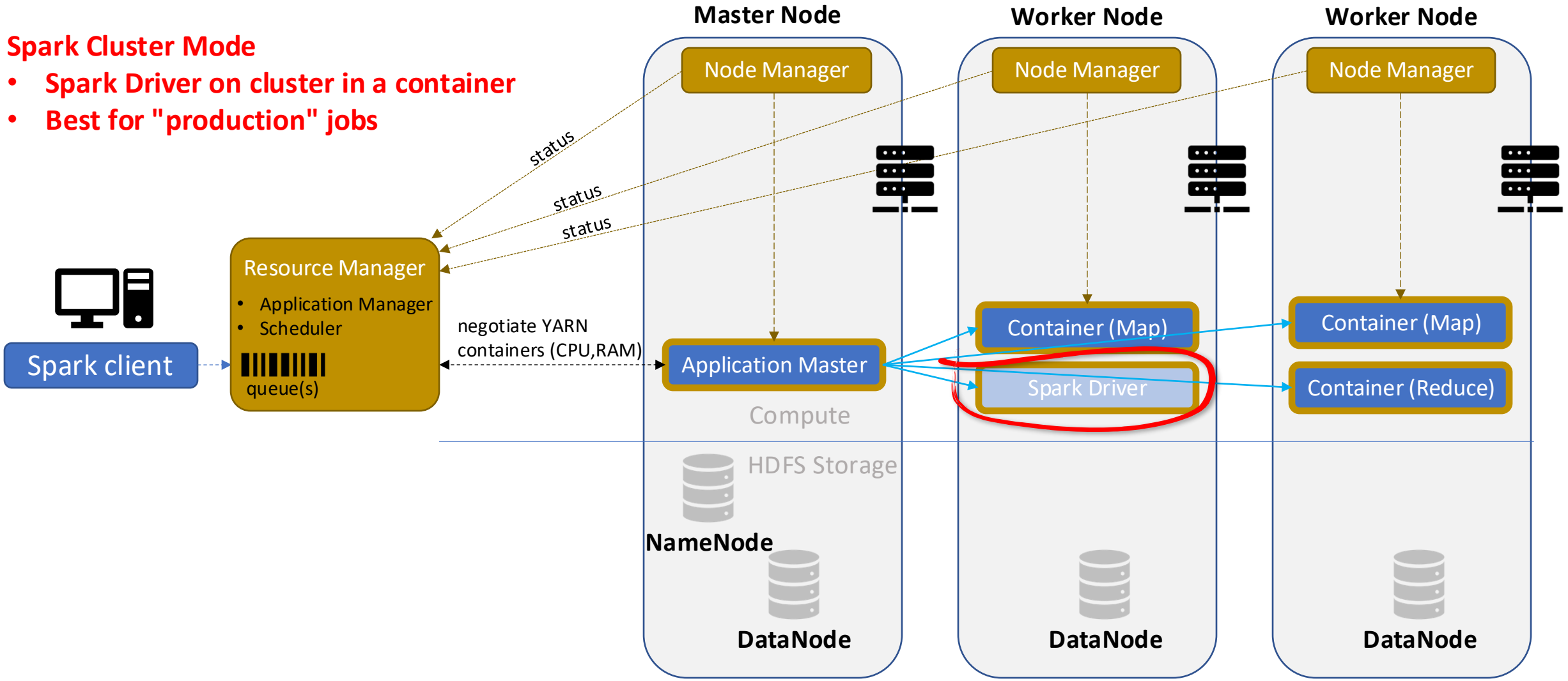
- **Spark Driver runs locally**
- **Best for interactive sessions**



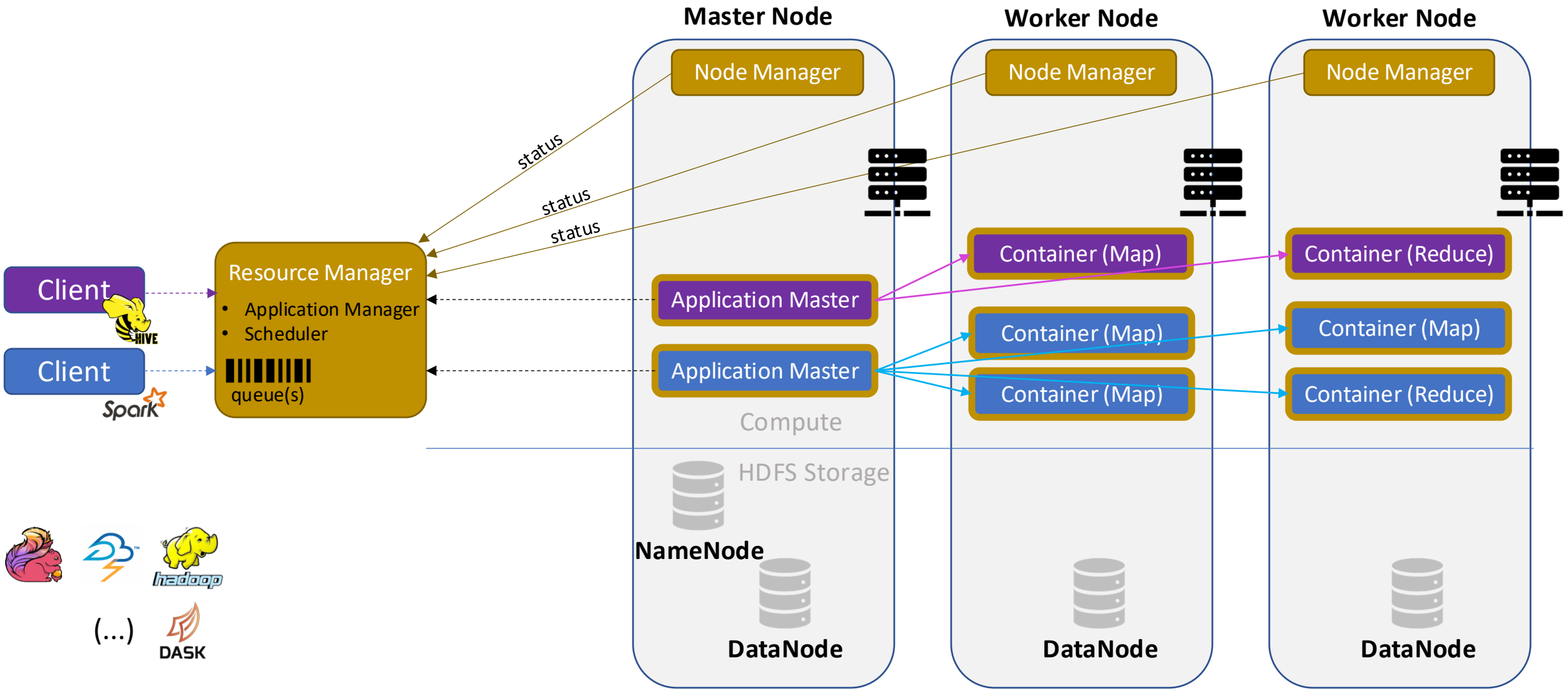
Spark on YARN

Spark Cluster Mode

- **Spark Driver on cluster in a container**
- **Best for "production" jobs**

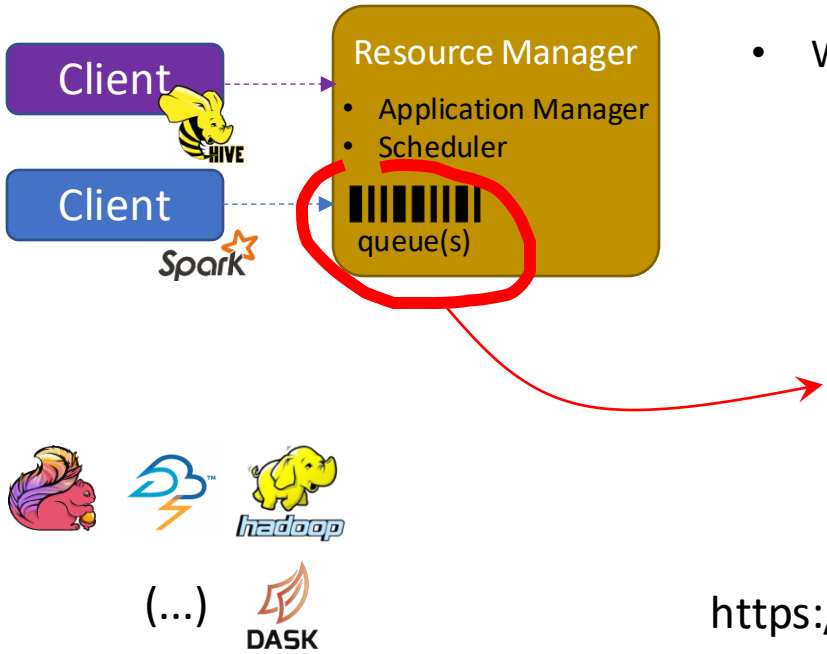


Spark on YARN

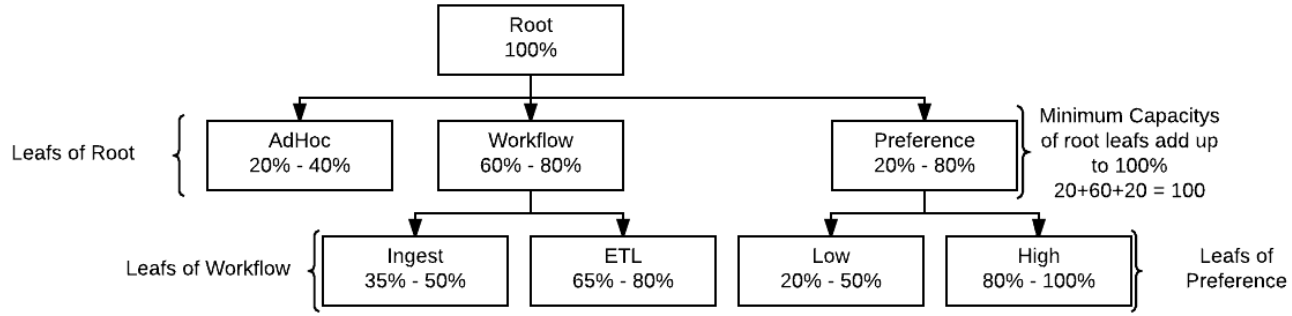


YARN – Capacity Scheduler

Not enough resources? You will be placed in a queue until resources become available.



- Queues are laid out in a hierarchical design, top most queue is the “root”
  - Resource reservations of a queue can be shared by child queues
- Child queues are given a min and max % of its parent reservation
  - Min: minimum % guaranteed when all resources are used
  - Max: maximum % allowed when resources are underused
- When an applications is submitted (enter its assigned queue):
  - It is deployed and run if the queue has enough reservation available for the application
  - Otherwise the application waits for the resources to become available



https://blog.cloudera.com/yarn-capacity-scheduler/

Module 2 – Self-evaluation

I know

- When to consider distributed data technologies versus (more) efficient single host solutions
- What is HDFS and purpose it serves, and what are comparable technologies (S3, etc.)
- What are what are warehouse or lakehouses (Trino, Hive), what they are used for
- What are NoSQL databases (Hbase, etc.) and what purpose they serve
- Most popular storage encodings: ORC, PARQUET, …
- Familiar with key words

I am able to

- Copy data to and from HDFS, explore HDFS
- Move data around in HDFS, modify access rights
- Create and manage data warehouses & lakehouses
- Optimization techniques (partitioning, efficient data format like ORC, PARQUET)
- Run ETL and OLAP queries

Next week – Module 3a

Pamela Delgado

Assignment 2

https://dslabgit.datascience.ch/course/2025/assignment-2

Fork in under your group and git clone

Start your engines

https://dslabgit.datascience.ch/course/2025/module-2c

(Fork and git clone)

Appendix

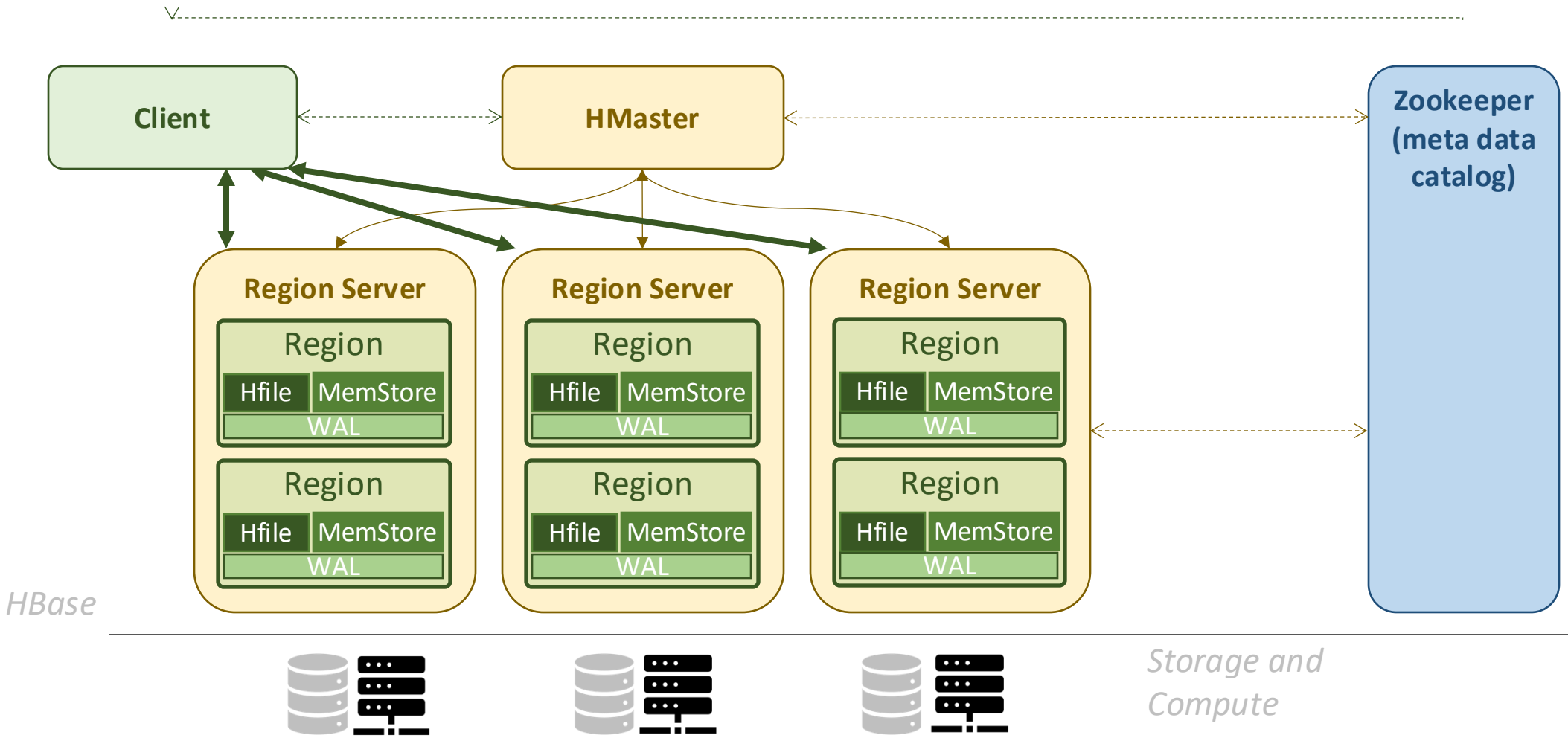
A1 - HBase Architecture

Knowing the Hbase terminology is handy if you ever need to troubleshot it ...

- **HRegion**: tables are split into multiple HRegion, or blocks of continuous data between start/end primary keys.
- **HRegionServer**: one per machine, manages a collection of HRegion on the big data cluster.
- **HMaster**: does not store data, but is responsible for managing the meta-data with the mapping of HRegions to HRegionServers. It's possible to have several HMaster, however only one at the time can be elected to orchestrate the work on the HRegionServer..

- **HStore**: underlying HBase storage consists of a memstore (in memory) and a store file (HFile) on disk.
- **HLog**: or Write Ahead Log file, is responsible for atomicity and durability, in case of HRegionServer failure, its HLog is split and distributed among the other HRegionServers for replay.

A1 - HBase Architecture



Hive SerDe – Serialization Deserialization

- Allows Hive to read in data from HDFS or a table, and write it back out to HDFS or other storage in any custom format.
- E.g.
  - JSON Format
  - HBase

A2 - JSON Format (Hive)

```
CREATE EXTERNAL TABLE  my_table(line STRING)
STORED AS TEXTFILE
LOCATION '/my/table/jsondoc';
```

```
SELECT  get_json_object(line, '$.first_name') AS first_name,
        get_json_object(line, '$.last_name') AS last_name
FROM my_table;
```

A2 - JSON SerDe (Hive)

```
CREATE EXTERNAL TABLE mytable(
        first_name STRING,
        last_name STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.JsonSerDe'
WITH SERDEPROPERTIES("ignore.malformed.json"="true")
STORED AS TEXTFILE
LOCATION '/my/table/jsondoc';
```

- Starting with Hive 4.0.0, you are able to use STORED AS JSONFILE

A3 - Hive / HBase SerDe

```
CREATE EXTERNAL TABLE hive_on_hbase(
                SensorID   STRING,
                humidity  STRING,
                temperature STRING,
                CO2 STRING
)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
   "hbase.columns.mapping"=":key, x:h, x:temp, y:co2"
)
TBLPROPERTIES(
   "hbase.table.name"="my_namespace:my_table",
   "hbase.mapred.output.outputtable"="my_namespace:my_table"
)
```

https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration

It is possible to use HBase as a source to HiveQL or Spark.

Example, to join Hive tables with HBase tables: :

- Given HBase table **my_namespace:my_table** with 2 column families **x** and **y**
- Table definition on left maps SensorID to the RowKeys of the hbase table, and other columns are mapped as follow:

  humidity -> x:h
  temperature -> x:temp
  CO2 -> y:co2

Note: In practice, you typically do not process the entire content of an HBase table. Instead, queries are designed to access subsets of data (with WHERE predicate).

A4 - User Defined Functions (UDF)

- Hive QL comes with many UDF by default
  - For a list, run the Hive query: SHOW FUNCTIONS;
  - Or find them in the documentation https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-DateFunctions
- You can also create your own UDF
  - Write a java plugin https://cwiki.apache.org/confluence/display/Hive/HivePlugins
  - Then add the JAR file to your Hive session and create a function
    - ADD JAR hdfs://<host>:<port>/<path>
    - CREATE TEMPORARY FUNCTION <FUNCTION_NAME> AS <JAVA_CLASS_NAME>
- The above steps can be used to enable ESRI spatial framework UDF (as of Hive 4.0.0 they are supported natively)