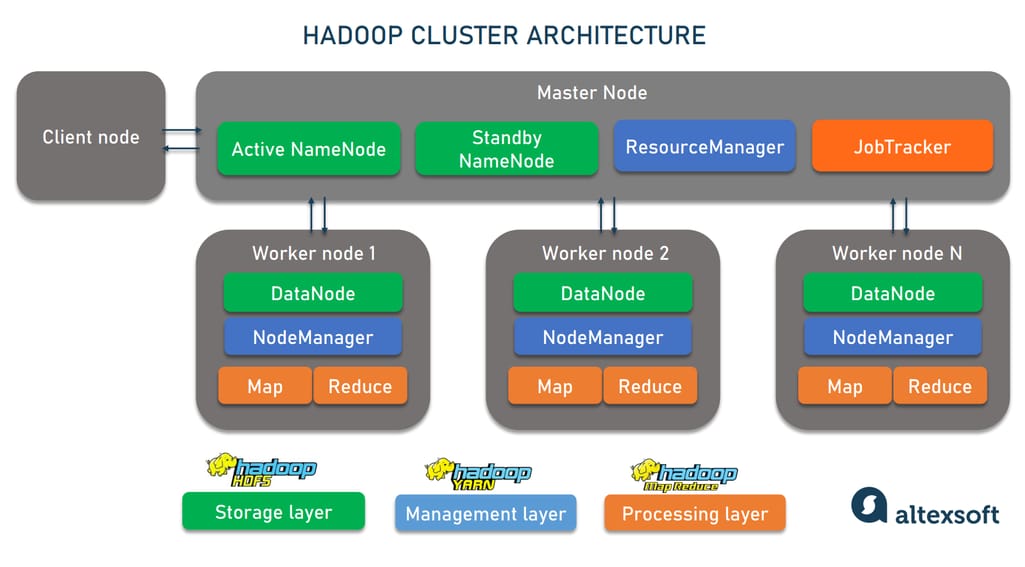
Q1 : How many versions Hadoop has?

**Major Versions:**

Three major versions of Apache Hadoop:

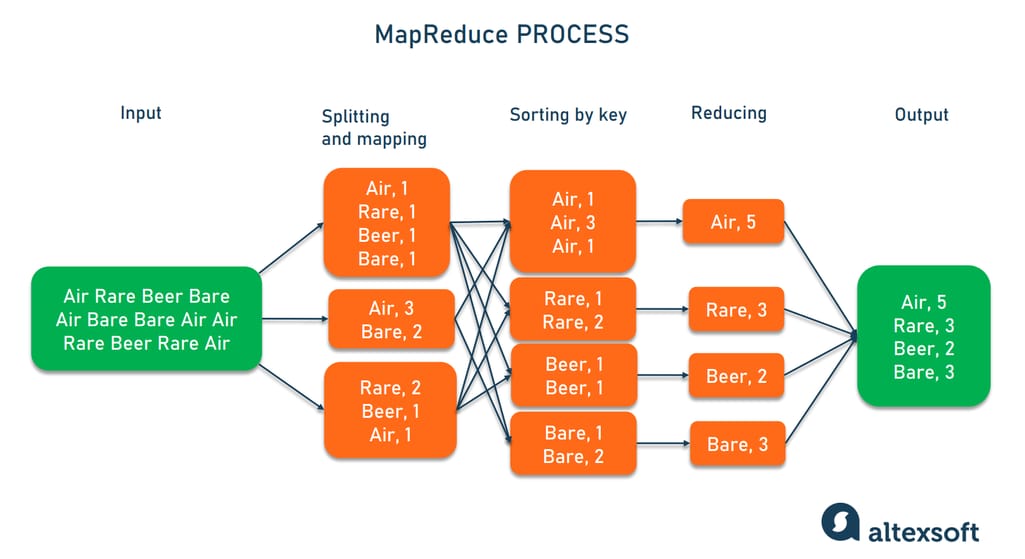
1. **Hadoop 1.x:** This was the initial stable release, with the core components being HDFS (Hadoop Distributed File System) and MapReduce. Hadoop 1.x had limitations in scalability, particularly with the JobTracker managing all jobs in the cluster. The latest release in this line was 1.2.1 (August 2013).
2. **Hadoop 2.x (YARN):** This major release introduced YARN (Yet Another Resource Negotiator), which significantly improved resource management and allowed Hadoop to support more than just MapReduce applications. YARN decoupled resource management from the MapReduce programming model. The Hadoop 2.x line has seen numerous minor and maintenance releases, with the latest being 2.10.2 (May 2022), which is still maintained.
3. **Hadoop 3.x:** This version introduced several significant enhancements, including support for Java 8+, erasure coding for storage efficiency, more than two Name Nodes for HDFS high availability, and a rewritten YARN resource manager for better scalability and federation. The Hadoop 3.x line is actively maintained, with the latest release being 3.4.1 (October 2024).

Q2 : A Data node Machine contains exactly?



* **Node Manager (NM):** Each machine in a YARN-based Hadoop cluster that is responsible for executing tasks (like MapReduce tasks) runs exactly one Node Manager. The Node Manager manages the resources (CPU, memory, disk, network) available on that machine and reports to the Resource Manager.
* **One Resource Manager (RM):** There is typically only **one active Resource Manager** in a Hadoop YARN cluster. The Resource Manager is responsible for allocating resources across the entire cluster. It doesn't reside on every DataNode machine.
* **One Application Master (AM):** An Application Master is specific to each running application in YARN (e.g., each MapReduce job has its own Application Master). Application Masters negotiate resources from the Resource Manager and work with the Node Managers to execute tasks. They are not a permanent component running on every DataNode machine.
* **One MapReduce job :** A MapReduce job is a specific computation. While parts of a MapReduce job (the Map and Reduce tasks) will run on DataNodes within containers managed by the Node Manager, a DataNode machine doesn't "contain" an entire MapReduce job. Many tasks from different jobs can run on a single DataNode over time.

Q3 : Which one of the following statements is true regarding a MapReduce job?



**Input to mapper is sorted key, value pairs:** The input to the mapper is typically raw data split into chunks. These chunks are then processed record by record, and the mapper transforms these records into key-value pairs. The input to the mapper is generally **not sorted**.

**Number of mapper tasks is equal to the number of reducer tasks:** The number of mapper tasks is primarily determined by the number of input splits of the data. The number of reducer tasks is configured independently based on the desired level of parallelism for the reduce phase and the number of output partitions. These two numbers are often different.

**Reducer starts only when all mapper finish execution:** The reduce phase can only begin after all the mapper tasks have completed their execution. This is because the reducers often need to process the intermediate key-value pairs produced by all the mappers to perform the aggregation or summarization. The framework needs to ensure that all intermediate output is available before the reduce phase starts.

Q4 : Which of the following statements is correct for an external table of hive?

In Hive, an external table is a way to define a schema over data that resides outside of the Hive warehouse directory. Instead of moving the data into Hive's managed storage, you essentially create a pointer to the external data source. This data source could be in HDFS at a location you specify, or even in other storage systems

CREATE EXTERNAL TABLE IF NOT EXISTS user\_profiles (

user\_id INT,

username STRING,

city STRING

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LOCATION '/user/data/user\_info/';

**Source Data gets deleted if external table is dropped:** dropping an external table only removes the metadata definition in Hive, leaving the actual data in its original location unaffected.

**Source Data does not get deleted if the external table is dropped:** This is the defining characteristic of an external table in Hive.

**It provides transaction capabilities like RDBMS:** While more recent versions of Hive (with ACID properties enabled) offer transaction capabilities for **managed (internal)** tables, external tables generally do not participate in these transactions. The underlying data source might or might not have transactional properties, but Hive's external table definition itself doesn't impose or manage transactions on the external data.

Q5: which is the default hive metastore?

When you set up Hive without explicitly configuring a database for the metastore, it defaults to using an embedded Apache Derby database.

* **Single User:** Derby in its embedded mode supports only one user at a time. This means you can only have one Hive CLI session open at any given moment. If you try to open another session, you'll likely encounter an error.
* **Local Storage:** The metadata is stored on the local file system where the Hive metastore service is running.
* **Not Suitable for Production:** Due to the single-user limitation, Derby is generally not recommended for production environments or multi-user setups.

For more robust and production-ready Hive deployments, it is highly recommended to configure the metastore to use a more scalable relational database like **MySQL**, PostgreSQL, Oracle, or MS SQL Server. These databases support multiple concurrent users and provide better reliability and performance for the metastore.

Q6 : Which among these can act as hive execution engine?

Hive is designed to be flexible and can leverage different execution engines to process its queries. The choice of execution engine can impact performance and resource utilization. The common execution engines that Hive can work with include:

* **MapReduce:** This was the original execution engine for Hive. HiveQL queries are translated into a series of MapReduce jobs that run on the Hadoop cluster. While reliable, MapReduce can be slower for certain types of queries due to its disk-based intermediate data storage.
* **Spark:** Apache Spark is a faster and more general-purpose distributed processing engine. When configured to use Spark as the execution engine, Hive can achieve significant performance improvements, especially for iterative algorithms and complex transformations, as Spark can keep intermediate data in memory.
* **TEZ:** Apache Tez is another execution engine built on top of YARN that aims to optimize data flow and reduce the number of MapReduce stages required for a query. It uses a more efficient data transfer mechanism and can lead to faster query execution compared to traditional MapReduce.

You can configure which execution engine Hive uses through its configuration settings. The choice often depends on the specific workload, performance requirements, and the available resources in the Hadoop cluster.

Q7: MongoDB can be used as backend database in which of following use cases?

MongoDB's flexible schema, scalability, and ability to handle diverse data types make it a suitable backend database for all the mentioned use cases:

* **Video gaming application:** MongoDB can efficiently handle the high write loads and real-time data updates common in gaming. It can store player profiles, game state, inventory, leaderboards, and more. Its scalability allows it to accommodate a growing player base.
* **Content management application:** MongoDB's document-oriented nature is well-suited for storing and managing unstructured or semi-structured content like articles, blog posts, user comments, and multimedia. Its flexible schema allows for easy evolution of content models.
* **E-commerce applications:** MongoDB can store product catalogs with varying attributes, customer profiles, shopping cart information, order history, and more. Its scalability can handle large product inventories and high transaction volumes.

While relational databases are also commonly used for these applications, MongoDB offers advantages in terms of flexibility and scalability for certain aspects of these use cases. The specific choice often depends on the particular requirements and priorities of the application.

Q8 : collection in mongoDB is

* A **database** is a container for collections.
* A **collection** is a group of related documents. This is analogous to a table in a relational database, but without a fixed schema.
* A **document** is a set of key-value pairs. This is the basic unit of data in MongoDB and is similar to a row in a relational database, but with a dynamic schema.

Therefore, a collection organizes multiple documents that typically have a similar structure or purpose within a MongoDB database.

Q9 : In Spark, RDD stands for

A Resilient Distributed Dataset (RDD) is a fundamental data structure in Apache Spark. It represents an **immutable**, **distributed** collection of data that is partitioned across the nodes of a Spark cluster. Here's a breakdown of the key characteristics:

* **Resilient:** RDDs are fault-tolerant. If a partition of an RDD is lost due to a node failure, Spark can automatically reconstruct that partition using the lineage of transformations that were applied to create it. This lineage, or the history of operations, is recorded for each RDD.
* **Distributed:** The data in an RDD is divided into multiple partitions, which can be stored and processed in parallel across the different nodes in a Spark cluster. This parallel processing is what enables Spark to achieve high performance on large datasets.
* **Dataset:** An RDD represents a collection of data elements. This data can be in various formats, such as text files, CSV files, databases, or even in-memory objects.
* **Immutable:** Once an RDD is created, it cannot be changed. When you perform an operation on an RDD, Spark creates a new RDD representing the result of that operation, rather than modifying the original RDD. This immutability simplifies fault tolerance and makes it easier to reason about data transformations.

**Key operations on RDDs fall into two categories:**

* **Transformations:** These operations create a new RDD from an existing one. Examples include map, filter, flatMap, groupByKey, and reduceByKey. Transformations are lazy, meaning they are not executed immediately but are recorded as a lineage graph.
* **Actions:** These operations trigger the computation on the RDD and return a value to the driver program or write data to external storage. Examples include collect, count, first, take, reduce, and saveAsTextFile.

Q10 : The various ways in which a Spark RDD can be persisted are

Spark RDDs can be persisted (cached) in various ways to avoid recomputing them in subsequent operations. Here are the common persistence levels:

* **Memory only (MEMORY\_ONLY):** This is the default persistence level. It stores the RDD as deserialized Java objects in the JVM memory. If the RDD doesn't fit entirely in memory, some partitions will not be cached and will be recomputed as needed. This offers the fastest performance but is limited by available memory.
* **Memory only serialized (MEMORY\_ONLY\_SER):** This level stores the RDD as serialized Java objects in the JVM memory. Serialization reduces the memory footprint compared to MEMORY\_ONLY, allowing more data to fit in memory. However, it incurs a higher CPU cost for serialization and deserialization during use.
* **Memory and hard drive (MEMORY\_AND\_DISK):** This level is a hybrid approach. It stores the RDD in memory (deserialized). If the RDD doesn't fit in memory, the excess partitions are spilled to disk. When needed, these partitions are read from disk. This provides a balance between memory usage and resilience to data not fitting in memory, but disk I/O is significantly slower than memory access.
* **Memory and hard drive serialized (MEMORY\_AND\_DISK\_SER):** Similar to MEMORY\_AND\_DISK, but the RDD is stored in serialized form both in memory (if it fits) and on disk. This minimizes both memory usage and disk space but has the highest CPU overhead due to serialization and deserialization.
* **Disk only (DISK\_ONLY):** This level persists the RDD only on disk. It offers the largest storage capacity but is the slowest due to all data being read from disk.
* **Off-heap (OFF\_HEAP):** This level stores the RDD in off-heap memory, which is memory managed outside the JVM heap. This can reduce garbage collection overhead and provide more predictable memory usage. It requires enabling off-heap memory in Spark.

Q1 :

**ALTER TABLE t1 RENAME TO t2**

* **UPDATE TABLENAME FROM t1 to t2** This syntax is not valid in Hive for renaming tables. The UPDATE command in Hive is used to modify data within a table, not to rename the table itself.
* **RENAME TABLE FROM t1 to t2** it's not the specific syntax used by Hive for renaming tables.
* **UPDATE TABLE t1 AS t2** This syntax is also not used for renaming tables in Hive. The AS keyword is typically used for creating aliases for tables within a query.

Q2 :

**db.users.find({}, {name: 1, \_id: 0}).sort({age: 1})**

**db.users.find({}, {name: 1, \_id: 0})**:

* + db.users.find(): This is the basic command to query the users collection.
  + The first argument {} is an empty query document. This means we want to retrieve all documents in the users collection.
  + The second argument {name: 1, \_id: 0} is a projection document.
    - name: 1: This specifies that we want to include the name field in the returned documents.
    - \_id: 0: This specifies that we want to exclude the default \_id field, which MongoDB automatically adds to each document. This is often done to get a cleaner output with only the desired fields.
* **.sort({age: 1})**:
  + .sort(): This method is used to sort the results of the find() operation.
  + {age: 1}: This specifies the sorting criteria.
    - age: This indicates that we want to sort based on the value of the age field.
    - 1: This indicates ascending order (from lowest age to highest age).
* **db.users.find().limit()**: This query would retrieve a limited number of documents from the users collection
* **db.users.find()**: This query would retrieve all documents from the users collection without any specific sorting or projection. It would display all fields, not just the name, and the order would be the natural order in which the documents are stored
* **db.users.find().sort({age: -1})**: This query would sort the users by age in **descending** order (from highest age to lowest age) because -1 indicates descending order. It also doesn't project only the name field, so it would display all fields.

Q3 :

**a) hadoop fs -put < local\_ directory\_path> <hdfs\_ directory\_path>**

* **hadoop fs**: This is the base command for interacting with the Hadoop Distributed File System.
* **-put**: This command is used to copy files or directories from the local file system to HDFS.
* **<local\_directory\_path>**: This should be the full path to the local file or directory you want to copy (in this case, the path to f1).
* **<hdfs\_directory\_path>**: This should be the full path to the destination directory in HDFS where you want to copy the file. If you want to copy the file with the same name into the HDFS directory, you would specify the HDFS directory path. If you want to rename the file in HDFS or specify a full path including the new filename, you would do that here.

**Why the other options are incorrect:**

* **hadoop fs -copyFromLocal <hdfs\_ directory\_path> < local\_ directory\_path>**: The -copyFromLocal command also copies files from the local file system to HDFS, but the order of the source and destination paths is reversed compared to what's needed to copy *to* HDFS. The local path should come first.
* **hadoop fs -copy < local\_ directory\_path> <hdfs\_ directory\_path>**: The -copy command is used for copying files or directories *within* the HDFS file system, not from the local file system to HDFS.
* **hadoop fs -move <hdfs\_ directory\_path> < local\_ directory\_path>**: The -move command is used to move files or directories *within* HDFS or from HDFS to the local file system. It removes the source file/directory after the move. This is not for copying from local to HDFS.

Q4 :

**df.withColumnRenamed(existingName, newName)**

The withColumnRenamed() method is specifically designed for renaming columns in a Spark DataFrame. It takes two arguments:

* existingName: A string representing the current name of the column you want to rename.
* newName: A string representing the desired new name for the column.

This method returns a *new* DataFrame with the specified column renamed. Spark DataFrames are immutable, so the original DataFrame df remains unchanged. You need to assign the result of withColumnRenamed() to a new DataFrame or overwrite the original one if you want to work with the renamed column.

* **df.withRenamed(existingName, newName)**: There is no built-in method in PySpark DataFrame API called withRenamed().
* **df.withColumn(existingName, newName)**: The withColumn() method is used to add a new column to a DataFrame or to replace an existing column with a new expression.
* **df.Renamed(existingName, newName)**: DataFrame objects in PySpark do not have a method called Renamed(). The correct method is withColumnRenamed().

Q5 :

Ways to convert an RDD into a DataFrame in PySpark.

* **spark.createDataFrame(rdd, schema=columns):** This is a common and explicit way to create a DataFrame from an RDD.
  + spark.createDataFrame(): This method, available from the SparkSession object, is specifically designed to create DataFrames.
  + rdd: This is the RDD you want to convert.
  + schema=columns
* **rdd.toDF(columns):** This is a more concise way to achieve the same result.
  + rdd.toDF(): This method is available directly on RDD objects. It converts the RDD into a DataFrame.
  + columns: Similar to option a, this allows you to specify the column names for the DataFrame. If you omit this, Spark will assign default column names (like \_1, \_2, etc.).
* **spark.convertDataFrame(rdd, schema = Columns):** There is no method called convertDataFrame on the SparkSession object.