1. **Explain Architecture of spark?**

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

It depends on two abstractions:

1. Resilient Distributed Dataset (RDD)

RDD Stands for:

Resilient: Restore the data on failure.

Distributed: Data is distributed among different nodes.

Dataset: Group of data.

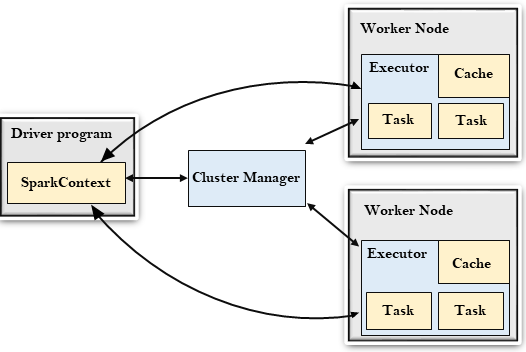
They are the group of data items that can be stored in-memory on worker nodes.

It enables you to recheck data in the event of a failure, and it acts as an interface for immutable data. It helps in recomputing data in case of failures, and it is a data structure. There are two methods for modifying RDDs: transformations and actions.

1. Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data.

The driver converts the program into a DAG for each job. A sequence of connection between nodes is referred to as a driver. As a result, you can read volumes of data using the Spark shell. You can also use the Spark context -cancel, run a job, task (work), and job (computation) to stop a job.



### The Spark driver

The master node (process) in a driver process coordinates workers and oversees the tasks. Spark is split into jobs and scheduled to be executed on executors in clusters. Spark contexts (gateways) are created by the driver to monitor the job working in a specific cluster and to connect to a Spark cluster. In the diagram, the driver programmes call the main application and create a spark context (acts as a gateway) that jointly monitors the job working in the cluster and connects to a Spark cluster. Everything is executed using the spark context.

Each Spark session has an entry in the Spark context. Spark drivers include more components to execute jobs in clusters, as well as cluster managers. Context acquires worker nodes to execute and store data as Spark clusters are connected to different types of cluster managers. When a process is executed in the cluster, the job is divided into stages with gain stages into scheduled tasks.

**The Spark executors**

An executor is responsible for executing a job and storing data in a cache at the outset. Executors first register with the driver programme at the beginning. These executors have a number of time slots to run the application concurrently. The executor runs the task when it has loaded data and they are removed in idle mode. The executor runs in the Java process when data is loaded and removed during the execution of the tasks. The executors are allocated dynamically and constantly added and removed during the execution of the tasks. A driver program monitors the executors during their performance. Users’ tasks are executed in the Java process.

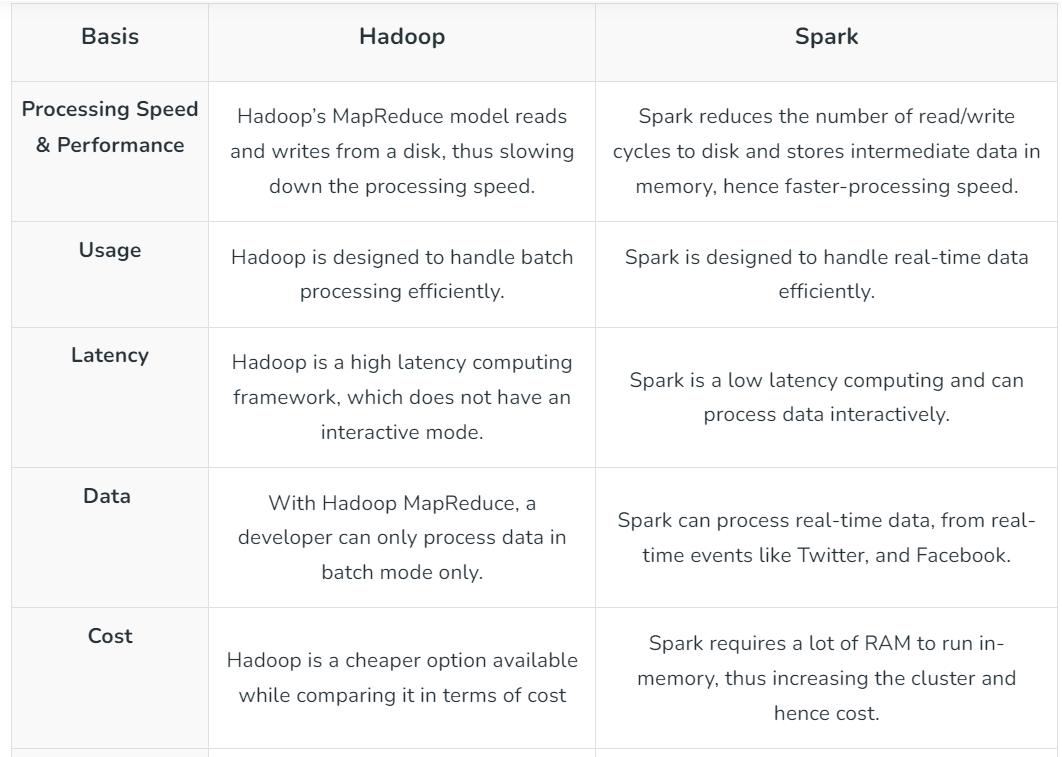
**Cluster Manager**

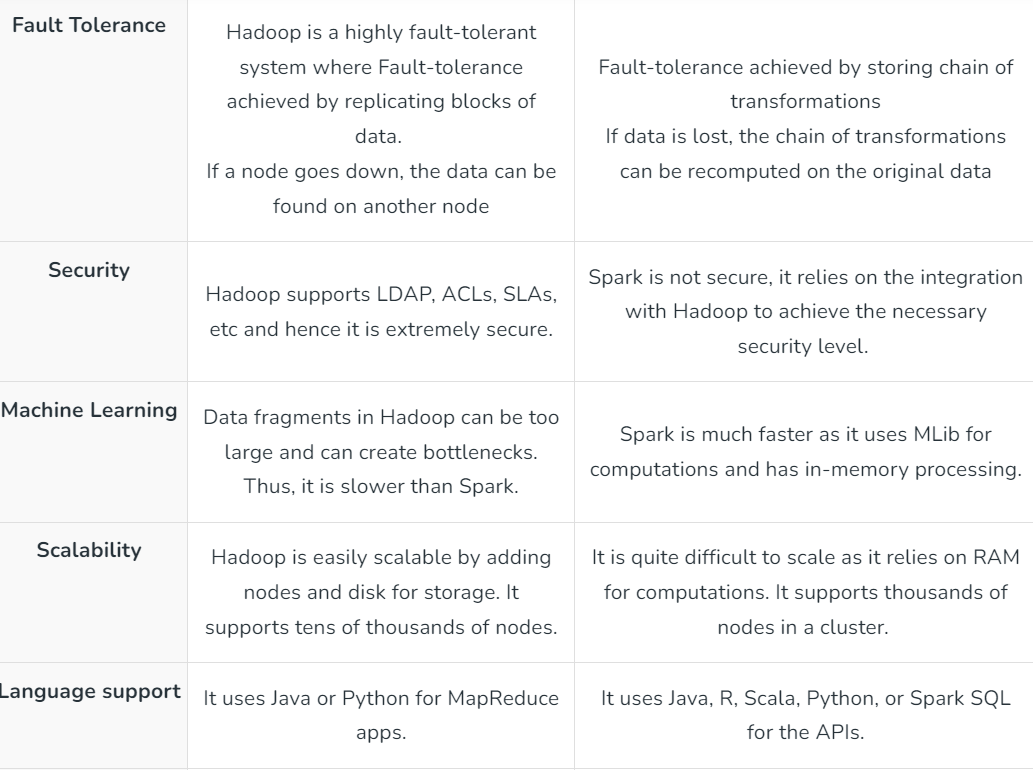
A driver program controls the execution of jobs and stores data in a cache. At the outset, executors register with the drivers. This executor has a number of time slots to run the application concurrently. Executors read and write external data in addition to servicing client requests. A job is executed when the executor has loaded data and they have been removed in the idle state. The executor is dynamically allocated, and it is constantly added and deleted depending on the duration of its use. A driver program monitors executors as they perform users’ tasks. Code is executed in the Java process when an executor executes a user’s task.

**Worker Nodes**

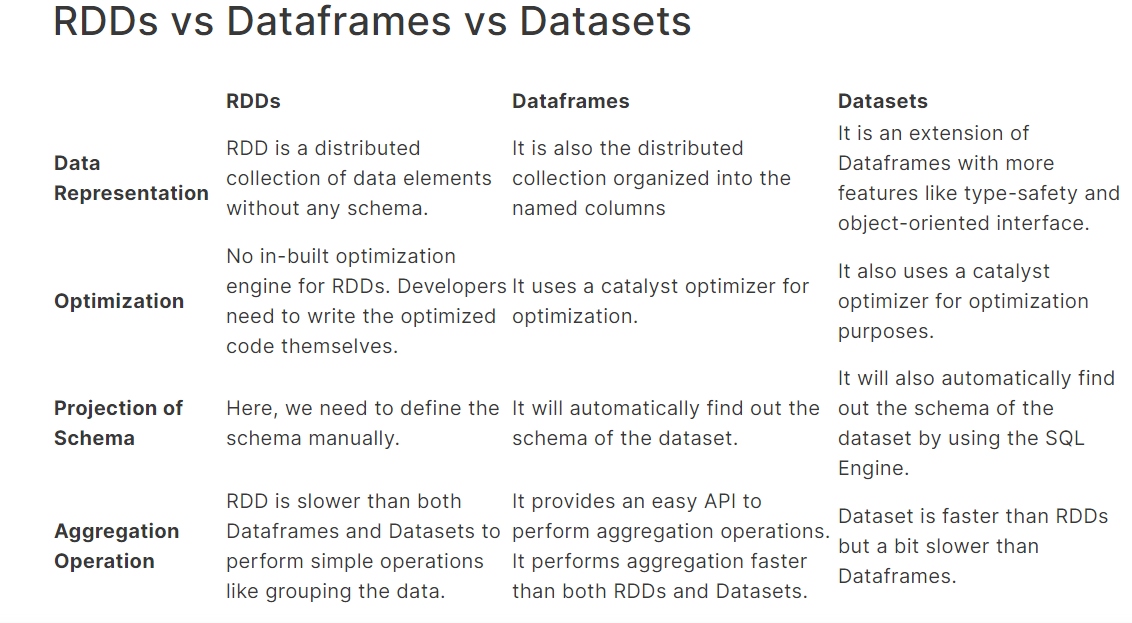
The slave nodes function as executors, processing tasks, and returning the results back to the spark context. The master node issues tasks to the Spark context and the worker nodes execute them. They make the process simpler by boosting the worker nodes (1 to n) to handle as many jobs as possible in parallel by dividing the job up into sub-jobs on multiple machines. A Spark worker monitors worker nodes to ensure that the computation is performed simply. Each worker node handles one Spark task. In Spark, a partition is a unit of work and is assigned to one executor for each one.

1. **Difference Between Hadoop and spark?**

****

****

1. **Difference Between RDD, Dataset and Dataframes?**

****

1. **Explain the similarities in all API of Spark?**
2. RDD API (Resilient Distributed Dataset):

Similarities:

- Immutability: RDDs are immutable, meaning their contents cannot be changed once they are created. Any transformation on an RDD creates a new RDD.

- Partitioning: RDDs are divided into partitions, which are the basic units of parallelism in Spark.

- Transformations and Actions: Like other Spark APIs, RDD API supports transformations (e.g., `map`, `filter`, `reduce`) and actions (e.g., `count`, `collect`, `saveAsTextFile`).

2. DataFrame API:

- Similarities:

- Immutability: Similar to RDDs, DataFrames are also immutable.

Any operation on a DataFrame creates a new DataFrame.

- Lazy Evaluation: Both DataFrame and RDD APIs support lazy evaluation, where transformations are not executed immediately but are evaluated only when an action is called.

- Operations: DataFrame API provides a higher-level abstraction that allows operations similar to SQL queries (e.g., `select`, `filter`, `groupBy`), making it more user-friendly for those familiar with SQL.

3. Dataset API:

- Similarities:

- Type Safety: Datasets bring type safety to Spark, similar to RDDs. This allows the use of strongly-typed objects and provides compile-time type checking.

- Immutability: Datasets, like RDDs and DataFrames, are also immutable in nature.

- Performance Optimization: Datasets share the same Catalyst query optimization engine with DataFrames, enabling optimizations during the query planning phase.

4. Spark Streaming API:

- Similarities:

- Micro-batching: Spark Streaming processes data in small, configurable batches, providing fault-tolerance and ease of integration with batch processing.

- Transformations: Both batch processing (using RDDs, DataFrames, or Datasets) and streaming processing (using DStreams) support similar transformations and actions.

1. **What is Transformations? Explain in Detail.**

RDD transformations are the methods that we apply to a dataset to create a new RDD. It will work on RDD and create a new RDD by applying transformation functions. The newly created RDDs are immutable in nature and can’t be changed.\

All transformations in Spark are lazy in nature that means when any transformation is applied to the RDD such as [map ()](https://www.cloudduggu.com/spark/transformations-actions/#map), [filter ()](https://www.cloudduggu.com/spark/transformations-actions/#filter), or [flatMap()](https://www.cloudduggu.com/spark/transformations-actions/" \l "flatMap), it does nothing and waits for actions and when actions like collect(), take(), foreach() invoke it does actual transformation/computation on the result of RDD.

Transformations are used to build a directed acyclic graph (DAG) of the computation, where each node represents a transformation, and edges represent dependencies between transformations. When an action is called, Spark uses the DAG to execute the transformations in the most efficient way.

Transformations are building blocks for constructing complex data processing pipelines in Spark. As mentioned earlier, transformations are not executed immediately; they are only triggered when an action, such as collect or saveAsTextFile, is called on the RDD, DataFrame, or Dataset.

1. **What is Actions in spark? Explain in Detail.**

In Apache Spark, actions are operations that trigger the execution of the computation plan (DAG - Directed Acyclic Graph) built using transformations. While transformations define the sequence of data processing steps, actions are operations that actually perform computations and produce a result. When an action is invoked, Spark schedules the execution of the transformations necessary to produce the result and initiates the computation on the distributed data.

Actions are crucial for executing the Spark computation plan and obtaining results. It's important to note that actions trigger the evaluation of the entire DAG of transformations, and Spark optimizes the execution plan to minimize data movement and maximize parallelism across the cluster.

Actions are the operations that return a value to the driver program or write data to an external storage system. Actions in Spark are eager, meaning they cause the evaluation of the transformations and materialize the result. Here are some common actions in Spark:

1. collect:

- Functionality: Retrieves all the elements of an RDD, DataFrame, or Dataset and brings them to the driver program.

- Example:

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

val collectedNumbers = numbers.collect()

2. count:

- Functionality: Returns the number of elements in an RDD, DataFrame, or Dataset.

- Example:

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

val count = numbers.count()

3. first:

- Functionality: Returns the first element of an RDD, DataFrame, or Dataset.

-Example:

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

val firstElement = numbers.first()

4. take:

- Functionality: Returns an array with the first n elements of an RDD, DataFrame, or Dataset.

- Example:

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

val firstThreeElements = numbers.take(3)

5. reduce:

- Functionality: Aggregates the elements of an RDD, DataFrame, or Dataset using a specified associative and commutative function.

- Example:

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

val sum = numbers.reduce((x, y) => x + y)

1. **What is Wide Transformation? Explain with example.**

In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.

**Wide transformations** involve the shuffling of data across partitions, which requires data to be reorganized and exchanged between different nodes in the cluster. This typically involves a stage boundary and can incur more overhead compared to narrow transformations. Wide transformations are often associated with operations that require data to be rearranged or grouped, leading to the need for data movement.

**Example**

val pairs = sc.parallelize(Seq(("apple", 1), ("orange", 2), ("apple", 3), ("orange", 4)))

// Applying wide transformation: groupByKey

val groupedByKey = pairs.groupByKey()

// Action to collect and print the result

groupedByKey.collect().foreach {

case (key, values) => println(s"$key: ${values.mkString(", ")}")

}

In this example, we have a pair RDD (pairs) with key-value pairs representing fruits and their corresponding quantities. The groupByKey transformation is a wide transformation because it requires data shuffling. It groups the values associated with each key together, resulting in a new RDD where each key is associated with an iterable collection of values.

In the collect action, when we print the result, you'll notice that the values for each key are grouped together, but the process involves shuffling and exchanging data between partitions. This can be an expensive operation in terms of performance.

1. **What is Narrow Transformation? Explain with example.**

In *Narrow transformation*, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. *Narrow transformations* are the result of *map(), filter().*

**Narrow transformations** are transformations where the input data for each partition is derived from a single partition of the parent RDD. These transformations do not require data shuffling or movement across partitions and can be executed in parallel on each partition independently.

**Example**

val numbers = sc.parallelize(Seq(1, 2, 3, 4, 5))

// Applying narrow transformation: map

val squaredNumbers = numbers.map(x => x \* x)

// Action to collect and print the result

squaredNumbers.collect().foreach(println)

In this example, the map transformation is a narrow transformation. Each element of the original RDD (numbers) is processed independently to produce a new RDD (squaredNumbers). The transformation applied to each element does not require knowledge of other elements in the RDD, and no data shuffling is involved. Each partition operates independently on its subset of the data.

1. **Write down the query of wide and narrow transformation with example?**

**Narrow Transformation Example (map)**

Narrow transformations involve operations where each partition of the resulting DataFrame can be computed independently, without requiring data to be shuffled or redistributed across partitions.

// Example DataFrame

val df = spark.createDataFrame(Seq(

("Alice", 34),

("Bob", 45),

("Charlie", 28)

)).toDF("Name", "Age")

// Narrow transformation: map

val nameLengthDF = df.map(row => (row.getString(0), row.getString(0).length))

// Action to display the result

nameLengthDF.show**()**

In this example, the `map` transformation is a narrow transformation. It computes the length of each name in the DataFrame independently without requiring data to be shuffled between partitions. Each partition processes its subset of the data in parallel, resulting in an efficient computation.

**Wide Transformation Example (groupBy)**

Wide transformations involve operations that require data to be shuffled or redistributed across partitions, typically requiring the exchange of data between nodes in the cluster.

// Example DataFrame

val df = spark.createDataFrame(Seq(

("Alice", "Engineering"),

("Bob", "Sales"),

("Charlie", "Engineering")

)).toDF("Name", "Department")

// Wide transformation: groupBy

val departmentCountDF = df.groupBy("Department").count()

// Action to display the result

departmentCountDF.show()

In this example, the `groupBy` transformation is a wide transformation. It involves shuffling the data across partitions to group records by the "Department" column. Data from different partitions needs to be exchanged to ensure that all records with the same department value are grouped together. Wide transformations like `groupBy` often involve a significant amount of network traffic and can impact performance, especially with large datasets.

1. **Explain Kerberos Architecture.**

Kerberos is a widely used authentication protocol that provides a secure way for users and systems to prove their identity in a networked environment. The Kerberos architecture involves multiple components working together to enable secure authentication and authorization.

1. User:

- The end user who wants to access a service or resource in the network.

- Users authenticate themselves to the Kerberos system by providing their credentials (usually a username and password).

2. Key Distribution Center (KDC):

- The central authority responsible for authentication and key distribution in the Kerberos system.

- Consists of two main components:

- Authentication Server (AS): Handles initial user authentication and issues Ticket Granting Ticket (TGT).

- Ticket Granting Server (TGS): Grants service tickets to users for accessing specific services.

3. Realm:

- A logical administrative domain in which the Kerberos authentication service operates.

- Realms have a unique name and trust relationship with other realms.

4. Authentication Process:

- When a user wants to access a service, the authentication process involves the following steps:

1. Request for TGT: The user sends a request to the AS for a TGT by providing their credentials.

2. TGT Issuance: If the credentials are valid, the AS issues a TGT encrypted with the user's secret key.

3. Service Ticket Request: The user presents the TGT to the TGS and requests a service ticket for the desired service.

4. Service Ticket Issuance: If the TGT is valid, the TGS issues a service ticket for the requested service.

5. Tickets:

- Tickets are a key component of Kerberos authentication.

- Ticket Granting Ticket (TGT): Issued by the AS and used to request service tickets.

- Service Ticket: Issued by the TGS for accessing a specific service.

6. Service Server:

- The server hosting the desired service that the user wants to access.

- Verifies the authenticity of the user by decrypting the service ticket.

7. Key Encryption Key (KEK):

- A secret key shared between the AS and the user.

- Used to encrypt the TGT, ensuring secure communication between the AS and the user.

8. Session Key:

- A temporary key generated by the TGS and shared between the user and the service server.

- Used for secure communication between the user and the service server.

9. Cross-Realm Authentication:

- Kerberos supports cross-realm authentication, enabling users from different realms to authenticate and access services in other realms.