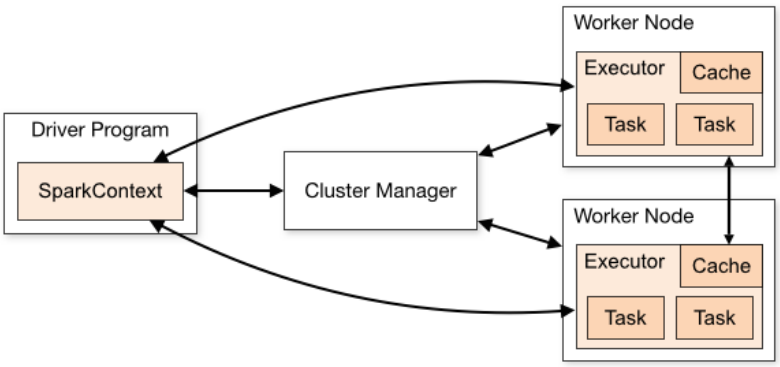
Test Q & A - Spark

1. Explain Architecture of Spark?



The Apache Spark framework uses a master-slave architecture that consists of a driver, which runs as a master node, and many executors that run across as worker nodes in the cluster.

When the Driver Program in the Apache Spark architecture executes, it calls the real program of an application and creates a SparkContext. SparkContext contains all of the basic functions. The Spark Driver includes several other components, including a DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager, all of which are responsible for translating user-written code into jobs that are actually executed on the cluster. Spark Driver and SparkContext collectively watch over the job execution within the cluster.

The Cluster Manager manages the execution of various jobs in the cluster. Spark Driver works in conjunction with the Cluster Manager to control the execution of various other jobs. The cluster Manager does the task of allocating resources for the job. Once the job has been broken down into smaller jobs, which are then distributed to worker nodes, SparkDriver will control the execution.

Many worker nodes can be used to process an RDD (Resilient Distributed Dataset) created in SparkContext, and the results can also be cached.

The SparkContext receives task information from the Cluster Manager and enqueues it on worker nodes. The executor is in charge of carrying out these duties. The lifespan of executors is the same as that of the Spark Application. We can increase the number of workers if we want to improve the performance of the system.

Apache Spark can be used for batch processing and real-time processing as well.

2. Difference between Hadoop & Spark

Hadoop and Spark are widely used open-source frameworks for big data architectures. The key differences between Hadoop and Spark are:

1. **Performance:** Spark is faster because it uses random access memory (RAM) instead of reading and writing intermediate data to disks. Hadoop stores data on multiple sources and processes it in batches via MapReduce.

2. **Cost:** Hadoop runs at a lower cost since it relies on any disk storage type for data processing. Spark runs at a higher cost because it relies on in-memory computations for real-time data processing, which requires it to use high quantities of RAM to spin up nodes.

3. **Processing:** Though both platforms process data in a distributed environment, Hadoop is ideal for batch processing and linear data processing. Spark is ideal for real-time processing and processing live unstructured data streams.

4. **Scalability:** When data volume rapidly grows, Hadoop quickly scales to accommodate the demand via Hadoop Distributed File System (HDFS). In turn, Spark relies on the fault tolerant HDFS for large volumes of data.

5. **Security:** Spark enhances security with authentication via shared secret or event logging, whereas Hadoop uses multiple authentication and access control methods. Though, overall, Hadoop is more secure, Spark can integrate with Hadoop to reach a higher security level.

6. **Machine learning (ML):** Spark is the superior platform in this category because it includes MLlib, which performs iterative in-memory ML computations. It also includes tools that perform regression, classification, persistence, pipeline construction, evaluation, etc.

3. Difference between RDD, DataFrame, Dataset

1) RDD (Resilient Distributed Dataset)

- RDD is the foundational data structure in Spark and represents a distributed collection of data, typically unstructured
- RDDs are inherently distributed and partitioned across the cluster.
- RDDs are immutable, and transformations on RDDs result in new RDD
- RDDs are low-level and do not have a schema
- RDDs offer full control and can handle complex data processing scenarios.

2) DataFrame

- DataFrame is a higher-level abstraction that represents distributed data in a tabular form with rows and columns.
- They have a schema, which defines the data types and names of columns.
- DataFrames are designed for structured data processing and optimization.
- DataFrame operations can be expressed using SQL-like syntax
- DataFrames can read from and write to various data sources

3) Datasets

- Dataset is a hybrid abstraction introduced in Spark that combines the best features of RDDs and DataFrames
- Datasets are strongly typed like RDDs but also have a well- defined schema like DataFrames.
- Datasets can work with both structured and unstructured data
- Datasets support both safe and unsafe type conversions, allowing you to handle data of different types flexibly.
- Datasets offer good interoperability with both Java and Scala, allowing developers to choose the language that best suits their needs.

4. Explain the similarities in all Spark APIs

Spark provides several APIs (such as DataFrame API, Dataset API, and Streaming API),) that developers can use to interact with the framework and perform data processing tasks. While these APIs have different purposes and syntax, they share some similarities in their overall design and functionality. Here are some common similarities across Spark APIs:

- **Distributed processing:** All Spark APIs are designed to take advantage of distributed computing resources. They enable parallel processing of data across a cluster of machines, allowing for efficient and scalable data processing.

- **Transformation and action operations:** Spark APIs provide transformations and actions to operate on RDDs or other data structures. Transformations are operations that generate a new RDD from an existing one, such as map, filter, or reduce ByKey. Actions trigger computations and return results to the driver program, for example, count, collect, or save.

- **Lazy evaluation:** Spark APIs employ lazy evaluation, meaning that transformations are not executed immediately. Instead, they build up a directed acyclic graph (DAG) of operations that are executed only when an action is called.

This optimization technique improves performance by avoiding unnecessary computations.

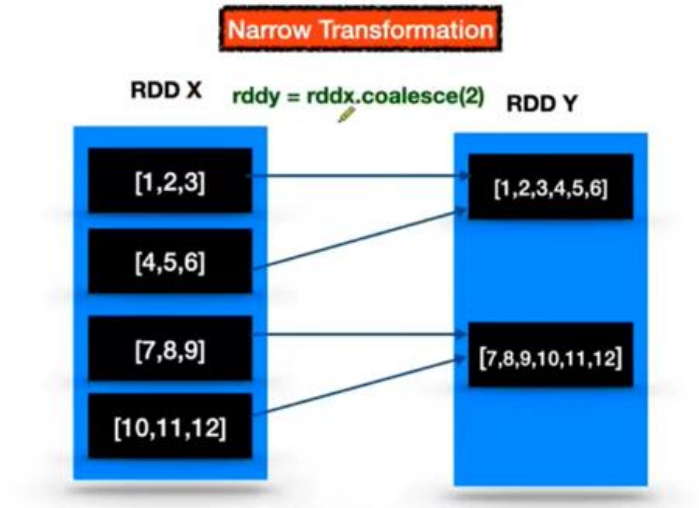
- **Data source and sink integration:** Spark APIs provide connectors to various data sources and sinks, allowing users to read data from different file formats (such as CSV, JSON, or Parquet) and databases (such as HDFS, Amazon S3, or Apache Cassandra) as well as write data to them. This flexibility enables seamless integration with existing data ecosystems.

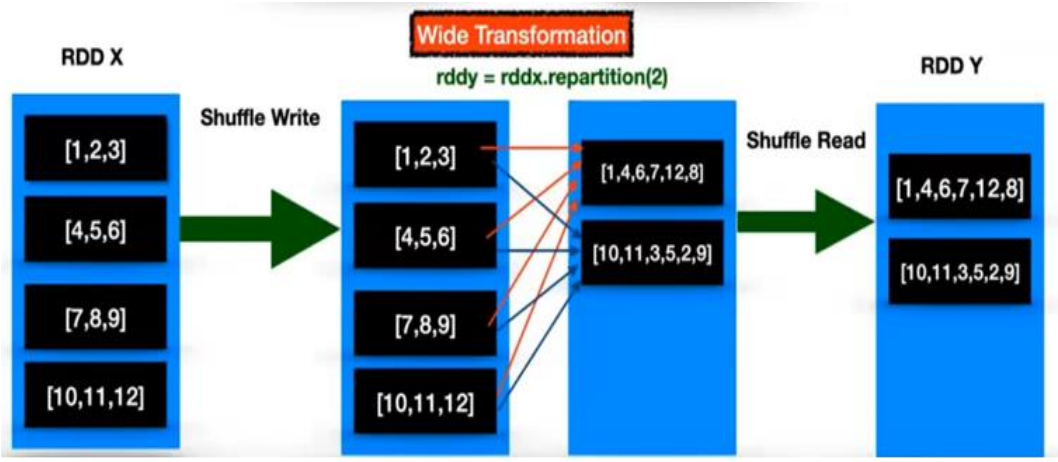
- **Language support:** Spark APIs are available in multiple programming languages, including Scala, Java, Python, and R. This language support allows developers to work with Spark using their preferred programming language, making it accessible to a broader range of users.

5. What is Transformation in Spark? 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(), filter(), or 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 can be categorized into **narrow transformations** and **wide transformations** based on their dependency on input partitions.





6. What is Actions in Spark? Explain in detail

In Spark, actions are operations that trigger the execution of transformations and return results or perform some side effects. Unlike transformations, which are lazily evaluated and create a new RDD, actions force the evaluation of RDDs and produce a final result or write data to an external system. Actions are the operations that drive the execution of the Spark job.

Actions can be categorized into two types based on the nature of their output:

1) Actions that return results

These actions perform computations on RDDs and return results to the driver program or write results to an external storage system. Examples of actions that return results include count(), collect(), take(), and reduce().

- count(): Returns the number of elements in the RDD.
- collect(): Retrieves all the elements of the RDD and returns them as an array to the driver program. This action should be used with caution when dealing with large datasets, as it collects all the data to the driver, potentially causing memory issues.
- take(n): Returns the first n elements from the RDD as an array.
- reduce(): Applies a binary function to reduce the elements of the RDD to a single result.

```
val rdd = sparkContext.parallelize(List(1, 2, 3, 4, 5))
val count = rdd.count()
val collectedData = rdd.collect()
val firstThreeElements = rdd.take(3)
val sum = rdd.reduce((a, b) => a + b)
```

2) Actions that perform side effects

These actions perform operations that have side effects, such as writing data to an external storage system or displaying output. Examples include saveAsTextFile(), foreach(), and foreachPartition().

- saveAsTextFile(path): Writes the elements of the RDD to a text file or files in the specified path.
- foreach(func): Applies a function to each element of the RDD. This action is useful for performing custom operations or side effects on each element.
- foreachPartition(func): Similar to foreach(), but applies the function to each partition of the RDD instead of individual elements. This action is useful when you need to perform operations that require partition-level processing or initialization.

7. What is the Wide Transformation? Explain with example

Wide transformations are transformations where each input partition can contribute to multiple output partitions. The resulting RDD may have a different number of partitions than the parent RDD. Wide transformations require shuffling or data movement across partitions, which introduces a data exchange between nodes. Examples of wide transformations include groupByKey (), reduceByKey (), and sortByKey ().

```
val rdd = sparkContext.parallelize(List(("apple", 3), ("banana", 2), ("apple", 5), ("banana", 1)))
val countRDD = rdd.reduceByKey((x, y) => x + y)
```

In this example, the reduceByKey () transformation is a wide transformation because it involves shuffling and merging the values for each key across partitions. The resulting RDD will have a different number of partitions than the parent RDD, depending on the number of unique keys.

8. What is Narrow Transformation? Explain with example

Narrow transformations are transformations where each input partition contributes to only one output partition. The resulting RDD has the same number of partitions as the parent RDD. Narrow transformations are executed in parallel on each partition of the parent RDD without shuffling or data movement across partitions. Examples of narrow transformations include map(), filter(), and union().

```
val rdd = sparkContext.parallelize(List(1, 2, 3, 4, 5))
val squaredRDD = rdd.map(x => x * x)
```

In this example, the map () transformation is a narrow transformation because each input partition can be independently processed to compute the squared value of each element. The resulting RDD will have the same number of partitions as the parent RDD.

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

Wide Transformation

Imagine a retail company that wants to perform customer segmentation based on purchasing behaviour. They have a large dataset of customer transactions, and they want to group customers based on their total purchase amount. This involves a wide transformation because it requires shuffling and aggregating data across partitions.

```
Input: Customer transactions dataset with columns (customer_id, purchase_amount)

Transformation:
val customerData = spark.read.csv("customer_transactions.csv")
val totalPurchase = customerData.groupBy("customer_id").sum("purchase_amount")

Output: Dataset with columns (customer_id, total_purchase_amount)
```

In this example, the groupBy() transformation is a wide transformation because it needs to shuffle and merge the purchase amounts for each customer across partitions to calculate the total purchase amount. This operation involves significant data movement and can be computationally expensive.

Narrow Transformation

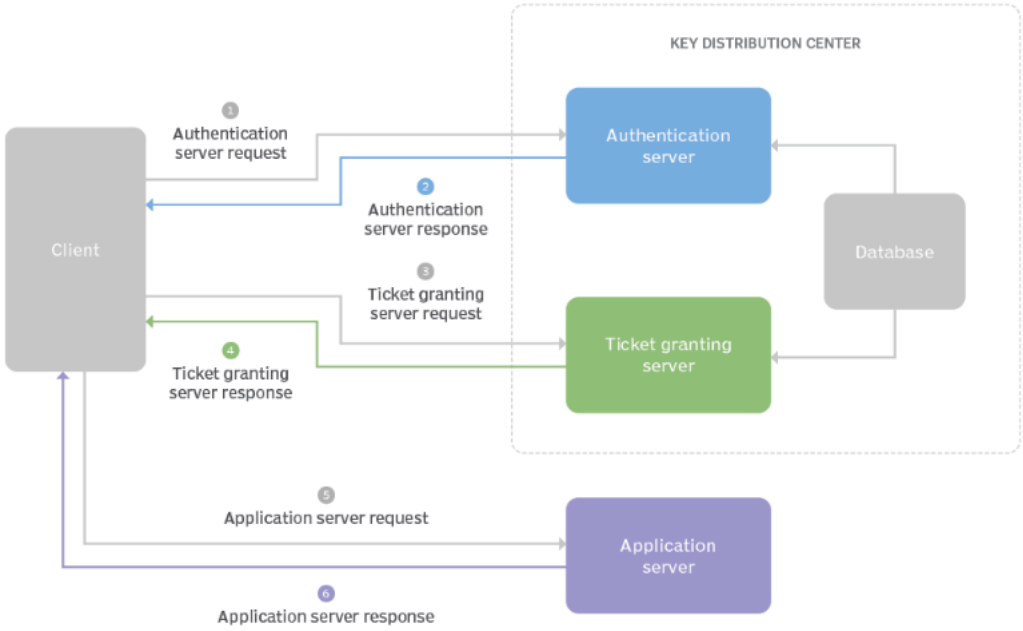
Consider a telecommunications company that wants to identify high-value customers based on their data usage. They have a dataset of customer records and want to filter out customers who have exceeded a certain data usage threshold. This involves a narrow transformation as each input partition contributes to only one output partition.

```
Input: Customer records dataset with columns (customer_id, data_usage)

Transformation:
val customerData = spark.read.csv("customer_records.csv")
val highValueCustomers = customerData.filter("data_usage > 1000")

Output: Dataset with filtered records of high-value customers
```

In this example, the filter() transformation is a narrow transformation because each input partition can independently evaluate the condition and contribute to the output partition. There is no need for data shuffling or merging, making it an efficient operation.

10. Explain Kerberos Architecture

Kerberos is a widely used authentication protocol that provides secure authentication for client-server applications over untrusted networks. It is based on symmetric key cryptography and operates using a client-server model.



The components and steps involved in the Kerberos architecture are:

Kerberos Components:

a. Kerberos Server (Key Distribution Center-KDC): The KDC is a central authority responsible for authenticating users and issuing security credentials. It consists of two main components:

- Authentication Server (AS): The AS verifies the user's identity during the initial authentication process and issues a Ticket Granting Ticket (TGT) if authentication succeeds.
- Ticket Granting Server (TGS): The TGS is responsible for issuing service tickets to users after they have been authenticated by the AS.

b. Client: The client is the user or entity requesting access to a service. It interacts with the KDC to obtain the necessary credentials for authentication.

c. Service Server: The service server hosts the services/resources that the client wants to access. It verifies the authenticity of the client using the service ticket.

d. Key Distribution Centre Database: The KDC maintains a database containing user and service principal information, along with their corresponding secret keys.

Kerberos Authentication Process:

The authentication process in Kerberos consists of the following steps:

a. Initial Authentication:

- The client sends a request to the KDC's AS component, requesting authentication for a particular service. The request includes the client's identity (username).
- The AS verifies the client's identity by checking its database. If the client is valid, the AS generates a session key (random symmetric key) and encrypts it using the client's password or secret key.
- The AS creates a Ticket Granting Ticket (TGT) containing the client's identity, the TGS's network address, and the encrypted session key. The TGT is encrypted using the TGS's secret key.
- The AS sends the TGT back to the client.

b. Ticket Granting:

- The client presents the TGT to the TGS, requesting a service ticket for the desired service.
- The TGS decrypts the TGT using its own secret key and verifies its authenticity.
- If the TGT is valid, the TGS generates a service ticket for the requested service. The service ticket contains the client's identity, the requested service's network address, and a copy of the session key encrypted with the service's secret key.
- The TGS sends the service ticket back to the client.

c. Service Authentication:

- The client presents the service ticket to the service server, along with a timestamp and other relevant information.
- The service server decrypts the service ticket using its secret key and verifies its authenticity.
- If the service ticket is valid, the service server grants the client access to the requested service.

Ticket Renewal and Session Management: After successful authentication, the client and service server establish a secure session using the shared session key. The session key is used to encrypt and decrypt subsequent communication between the client and the service server. To ensure security and prevent unauthorized access, Kerberos employs mechanisms for ticket renewal and session termination.

Ticket Renewal: The client can request ticket renewal by presenting the TGT to the KDC's TGS. The TGS verifies the TGT's validity and issues a renewed ticket if appropriate.

Session Termination: The client and service server can terminate the session by destroying the session key. This ensures that subsequent requests require re-authentication.

The Kerberos architecture provides a secure and efficient method for authentication and secure communication between clients and service servers. It addresses common security concerns, such as authentication, confidentiality, and integrity, by using encryption and trusted third-party authentication.
