Unit 4

1. **Apache Spark: Introduction and Core Concepts (15 marks)**

- **Definition**: Apache Spark is an open-source, distributed computing framework designed for big data processing and analytics, known for its speed and ease of use.
- **Core Features**:
  - **In-memory Computing**: Processes data in RAM, significantly faster than disk-based systems like Hadoop MapReduce.
  - **Fault Tolerance**: Uses Resilient Distributed Datasets (RDDs) to recover data automatically in case of node failures.
  - **Scalability**: Scales horizontally across clusters, handling petabytes of data.
- **Unified Engine**: Supports diverse workloads like batch processing, streaming, machine learning, and SQL queries within a single platform.
- **Ease of Use**: Provides APIs in Scala, Java, Python, and R, with high-level abstractions like DataFrames and Datasets.
- **Deployment Modes**: Runs on local machines, standalone clusters, or cloud platforms like AWS, Azure, and GCP.
- **RDDs (Resilient Distributed Datasets)**:
  - Fundamental data structure, representing immutable, partitioned collections of objects.
  - Supports operations like map, filter, and reduce for parallel processing.
- **Lazy Evaluation**: Optimizes execution by building a DAG (Directed Acyclic Graph) and evaluating transformations only when an action is triggered.

Spark Ecosystem and Components (15 marks)

- **Spark Core**: The underlying engine that manages task scheduling, memory management, and fault recovery, built around RDDs.
- **Spark SQL**:
  - Enables structured data processing using SQL queries or DataFrame APIs.
  - Integrates with Hive, JDBC, and other data sources for seamless querying.
- **Spark Streaming**:
  - Processes real-time data streams with micro-batch processing.
  - Supports integration with Kafka, Flume, and other streaming sources.
- **MLlib**:
  - Machine learning library offering scalable algorithms for classification, regression, clustering, and recommendation systems.
  - Includes tools for feature engineering and model evaluation.
- **GraphX**:
  - Library for graph processing and analytics, enabling computations on large-scale graph structures.
  - Supports algorithms like PageRank and connected components.
- **SparkR**: Extends Spark for R users, enabling data processing and machine learning with R syntax.
- **Integration with Big Data Tools**:

- Works with Hadoop HDFS, YARN, Cassandra, HBase, and cloud storage systems.
    - Leverages tools like Apache Hive for metadata management and Apache Kafka for streaming.
- **Cluster Managers**:
    - Supports multiple cluster managers like Apache YARN, Mesos, and Spark's standalone manager.
    - Ensures resource allocation and task distribution across nodes.

2. **Spark vs. Hadoop MapReduce (15 marks)**

- **Processing Model**:
    - **Spark**: Uses in-memory computing, storing intermediate data in RAM, which reduces I/O overhead and speeds up processing.
    - **Hadoop MapReduce**: Relies on disk-based processing, writing intermediate results to disk, leading to higher latency.
- **Performance**:
    - **Spark**: Up to 100x faster for iterative workloads (e.g., machine learning) due to in-memory processing.
    - **Hadoop MapReduce**: Slower for iterative tasks due to repeated disk I/O operations.
- **Ease of Use**:
    - **Spark**: Offers high-level APIs in Scala, Java, Python, and R, with abstractions like DataFrames and Datasets for simpler coding.
    - **Hadoop MapReduce**: Requires complex Java code with low-level Map and Reduce functions, making development more time-consuming.
- **Data Processing**:
    - **Spark**: Supports batch processing, real-time streaming, SQL queries, and machine learning within a unified engine.
    - **Hadoop MapReduce**: Primarily designed for batch processing, with limited support for other paradigms.
- **Fault Tolerance**:
    - **Spark**: Achieves fault tolerance through RDD lineage, recomputing lost partitions based on DAG.
    - **Hadoop MapReduce**: Uses data replication in HDFS for fault tolerance, which is robust but resource-intensive.
- **Scalability**:
    - Both scale horizontally across clusters, but Spark's in-memory model requires more RAM, while MapReduce is more disk-efficient.
- **Ecosystem Integration**:
    - **Spark**: Integrates with Hadoop HDFS, YARN, Hive, and other big data tools, offering a broader ecosystem (Spark SQL, MLlib, etc.).
    - **Hadoop MapReduce**: Works within the Hadoop ecosystem but lacks Spark's diverse libraries for streaming or machine learning.
- **Use Cases**:
    - **Spark**: Ideal for iterative algorithms, real-time analytics, and interactive data processing.

- **Hadoop MapReduce**: Suited for one-pass ETL jobs and batch processing of large datasets.

3. **Setting Up Spark Environment (15 marks)**

Prerequisites:

a. **Java**: Install Java Development Kit (JDK) 8 or later, as Spark requires Java. Verify with java -version.
b. **Scala**: Install Scala (version compatible with Spark, e.g., 2.12.x), as Spark is written in Scala. Verify with scala -version.
c. **Python**: Optional for PySpark; install Python 3.6+ and verify with python --version.

Downloading Spark:

d. Visit the official Apache Spark website (https://spark.apache.org/downloads.html).
e. Choose the latest stable version (e.g., Spark 3.5.x) and select a Hadoop version (e.g., Hadoop 3.2).
f. Download the .tgz file and extract it using tar -xzf spark-<version>-bin-hadoop<version>.tgz.

Setting Environment Variables:

g. Set SPARK_HOME to the extracted Spark directory (e.g., export SPARK_HOME=/path/to/spark-<version>-bin-hadoop<version>).
h. Add Spark's bin directory to PATH (e.g., export PATH=$SPARK_HOME/bin:$PATH).
i. Optionally, set JAVA_HOME to the JDK installation path (e.g., export JAVA_HOME=/path/to/jdk).

Local Installation:

j. Navigate to SPARK_HOME and test Spark shell with spark-shell (Scala) or pyspark (Python).
k. Verify the Spark UI at http://localhost:4040 to confirm the environment is running.

Cluster Setup (Optional):

l. **Standalone Mode**: Configure conf/spark-env.sh with master and worker settings, then start with ./sbin/start-all.sh.
m. **YARN/Mesos**: Configure conf/spark-defaults.conf with cluster manager details and submit jobs using spark-submit.

Dependency Management:

n. For Python, install PySpark via pip install pyspark==<version> to match the Spark version.
o. Ensure compatible Hadoop libraries if integrating with HDFS or YARN.

Testing the Setup:

p. Run a sample Spark job, e.g., spark-submit --class org.apache.spark.examples.SparkPi $SPARK_HOME/examples/jars/spark-examples_<version>.jar.
q. Check logs in $SPARK_HOME/logs for errors and validate output.

Cloud Setup (Optional):

r. Use managed services like AWS EMR, Google Dataproc, or Azure HDInsight for pre-configured Spark clusters.

    s. Configure access keys and network settings as per the cloud provider's documentation.

4. **RDDs and DataFrames (15 marks)**

- **RDDs (Resilient Distributed Datasets)**:
    - **Definition**: Fundamental data structure in Spark, representing an immutable, partitioned collection of objects distributed across a cluster.
    - **Resilience**: Achieves fault tolerance through lineage, recomputing lost partitions using a DAG (Directed Acyclic Graph).
    - **Operations**:
        - **Transformations**: Lazy operations like map, filter, and reduceByKey that create new RDDs.
        - **Actions**: Trigger computation, e.g., collect, count, and saveAsTextFile.
    - **Use Cases**: Low-level control for custom data processing, unstructured data, or when fine-grained transformations are needed.
    - **Performance**: S PIECE OF TEXT HERE
    - **Limitations**: Lacks schema information, requiring manual optimization for performance.
- **DataFrames**:
    - **Definition**: A distributed collection of data organized into named columns, similar to a relational database table, built on top of RDDs.
    - **Features**:
        - Supports SQL-like queries using Spark SQL for structured data processing.
        - Optimized by Catalyst optimizer for efficient query execution.
        - Integrates with DataSources like JSON, Parquet, and JDBC.
    - **Operations**: Supports DataFrame API methods like select, filter, groupBy, and join for declarative programming.
    - **Use Cases**: Ideal for structured/semi-structured data, ETL pipelines, and analytics with SQL queries.
    - **Performance**: Automatically optimized by Spark's query planner, reducing manual tuning compared to RDDs.
- **Comparison**:
    - **Abstraction Level**: RDDs are low-level, offering flexibility but requiring more coding; DataFrames are high-level, simpler for structured data.
    - **Optimization**: DataFrames leverage Catalyst optimizer; RDDs rely on user-defined logic.
    - **Ease of Use**: DataFrames are more user-friendly with SQL and API syntax; RDDs need functional programming expertise.
    - **Interoperability**: DataFrames can be converted to RDDs using .rdd and vice versa for hybrid workflows.

5. **Spark Core and Spark SQL (15 marks)**

- **Spark Core**:

    - **Definition**: The foundational engine of Apache Spark, providing the basic functionality for distributed data processing and task execution.
    - **Key Components**:
        - **RDDs (Resilient Distributed Datasets)**: Core data structure for fault-tolerant, parallel processing of data.
        - **DAG Scheduler**: Creates a Directed Acyclic Graph (DAG) to optimize and schedule tasks across the cluster.
        - **Task Scheduler**: Distributes tasks to worker nodes for execution.
    - **Functionality**:
        - Manages memory, fault recovery, and task coordination.
        - Supports transformations (e.g., map, filter) and actions (e.g., collect, count) on RDDs.
    - **Cluster Management**: Integrates with standalone, YARN, or Mesos for resource allocation.
    - **Use Cases**: Low-level data processing, custom algorithms, and unstructured data manipulation.
- **Spark SQL**:
    - **Definition**: A Spark module for structured and semi-structured data processing, enabling SQL queries and DataFrame/Dataset APIs.
    - **Key Features**:
        - **DataFrame API**: Provides a tabular abstraction for data manipulation with named columns.
        - **Catalyst Optimizer**: Automatically optimizes query plans for better performance.
        - **Hive Integration**: Supports Hive metastore, allowing queries on existing Hive tables.
    - **Functionality**:
        - Executes SQL queries directly or programmatically via DataFrame operations (e.g., select, groupBy, join).
        - Reads/writes data from formats like JSON, Parquet, ORC, and JDBC sources.
        - Supports User-Defined Functions (UDFs) for custom logic.
    - **Use Cases**: ETL pipelines, data warehousing, and ad-hoc analytics on structured data.
- **Relationship**:
    - Spark SQL is built on Spark Core, leveraging its distributed computing capabilities.
    - DataFrames in Spark SQL are abstractions over RDDs, combining ease of use with Core's fault tolerance and scalability.
    - Spark Core handles execution, while Spark SQL optimizes query processing for structured data.

6. **Spark Core Concepts (15 marks)**

- **Resilient Distributed Datasets (RDDs)**:
    - Immutable, partitioned collections of data distributed across a cluster.
    - Fault-tolerant through lineage, allowing recomputation of lost partitions.

    - Supports transformations (e.g., map, filter) and actions (e.g., count, collect).
- **Directed Acyclic Graph (DAG)**:
    - Represents the sequence of transformations on RDDs for lazy evaluation.
    - Optimizes execution by grouping operations and minimizing data shuffling.
- **Lazy Evaluation**:
    - Transformations are not executed immediately; they build a DAG until an action triggers computation.
    - Improves performance by optimizing the execution plan.
- **Task Scheduling**:
    - Spark's Task Scheduler assigns tasks to worker nodes based on data locality.
    - Works with cluster managers (e.g., YARN, Mesos, Standalone) for resource allocation.
- **Memory Management**:
    - In-memory computing stores data in RAM for faster processing.
    - Supports caching/persisting RDDs using cache() or persist() for iterative workloads.
- **Fault Tolerance**:
    - Achieved through RDD lineage, which tracks transformations to recompute lost data.
    - No need for data replication, unlike Hadoop HDFS.
- **Cluster Architecture**:
    - **Driver**: Runs the main program, creates SparkContext, and coordinates tasks.
    - **Executors**: Worker processes on nodes that execute tasks and store data.
    - **Cluster Manager**: Allocates resources (e.g., YARN, Mesos, or Spark Standalone).
- **Shuffle Operations**:
    - Data movement across nodes during operations like groupByKey or join.
    - Performance-intensive, optimized by minimizing shuffles through proper partitioning.

7. **Transformations and Actions in Spark (15 marks)**

- **Transformations**:
    - **Definition**: Operations that create a new RDD or DataFrame from an existing one, defining how data should be transformed.
    - **Lazy Evaluation**: Transformations are not executed immediately; they build a DAG until an action is triggered.
    - **Types**:
        - **Narrow Transformations**: Operations where each input partition contributes to one output partition (e.g., map, filter).
        - **Wide Transformations**: Operations requiring data shuffling across partitions (e.g., groupByKey, join).

- **Examples**:
  - map(func): Applies a function to each element, returning a new RDD.
  - filter(func): Returns a new RDD with elements passing the function.
  - reduceByKey(func): Aggregates key-value pairs by key using a function.
  - join(otherRDD): Combines two RDDs based on common keys.
- **Use Case**: Used to define data processing logic, such as cleaning, aggregating, or reshaping data.

- **Actions**:
  - **Definition**: Operations that trigger the execution of transformations and return results to the driver or write to storage.
  - **Immediate Execution**: Unlike transformations, actions initiate computation of the DAG.
  - **Examples**:
    - collect(): Retrieves all elements of an RDD/DataFrame to the driver.
    - count(): Returns the number of elements in an RDD/DataFrame.
    - saveAsTextFile(path): Writes RDD data to a text file in the specified path.
    - take(n): Returns the first n elements of an RDD to the driver.
  - **Use Case**: Used to obtain final results, save output, or inspect data after transformations.
- **Key Differences**:
  - **Execution**: Transformations are lazy; actions trigger computation.
  - **Output**: Transformations produce new RDDs/DataFrames; actions return non-RDD results or write data.
  - **Performance**: Minimize actions to reduce computation, as each action triggers a full DAG execution.
- **Practical Notes**:
  - Combine transformations to reduce shuffles (e.g., use reduceByKey instead of groupByKey for efficiency).
  - Avoid overusing actions like collect() on large datasets to prevent driver memory issues.

8. Introduction to Spark SQL (15 marks)

- **Definition**: Spark SQL is a module in Apache Spark for processing structured and semi-structured data, enabling SQL queries and programmatic DataFrame/Dataset APIs.
- **Purpose**: Simplifies data analysis by combining SQL's declarative querying with Spark's distributed computing capabilities.
- **Key Components**:
  - **DataFrame API**: Represents data as tables with named columns, similar to relational databases.

    - **Dataset API**: A type-safe extension of DataFrames (mainly in Scala/Java), combining RDD-like flexibility with SQL optimization.
    - **Catalyst Optimizer**: Automatically optimizes query plans for efficient execution.
- **Features**:
    - Supports standard SQL queries for data manipulation and aggregation.
    - Integrates with data sources like JSON, Parquet, ORC, CSV, JDBC, and Hive.
    - Enables User-Defined Functions (UDFs) for custom processing.
- **Architecture**:
    - Built on Spark Core, leveraging its RDDs and distributed computing.
    - Uses a unified engine to process SQL queries alongside other Spark workloads (e.g., streaming, ML).
- **Benefits**:
    - **Ease of Use**: Familiar SQL syntax for analysts and developers.
    - **Performance**: Catalyst Optimizer and Tungsten engine improve query execution speed.
    - **Interoperability**: Seamlessly works with Hive metastore and other big data tools.
- **Use Cases**:
    - ETL (Extract, Transform, Load) pipelines for data warehousing.
    - Ad-hoc querying and reporting on large datasets.
    - Combining SQL with machine learning or streaming workflows.
- **Basic Workflow**:
    - Create a SparkSession as the entry point for Spark SQL.
    - Load data into DataFrames from various sources.
    - Run SQL queries or DataFrame operations, then save or display results.

9. **Advanced Spark Programming (15 marks)**

- **Custom Partitioning**:
    - Control data distribution across nodes using custom partitioners to optimize performance.
    - Example: Use partitionBy with a custom Partitioner for key-based data locality.
- **Broadcast Variables**:
    - Share read-only data efficiently across nodes to avoid redundant data transfer.
    - Example: Broadcast a lookup table using sc.broadcast() for joins or filtering.
- **Accumulators**:
    - Distributed counters for aggregating information (e.g., error counts) across executors.
    - Example: Use sc.accumulator() to track invalid records during processing.
- **Performance Tuning**:

    - **Caching/Persisting**: Use cache() or persist() with appropriate storage levels (e.g., MEMORY_AND_DISK) for iterative computations.
    - **Shuffle Optimization**: Minimize shuffles by using reduceByKey over groupByKey and adjusting spark.sql.shuffle.partitions.
    - **Data Skew Handling**: Address uneven data distribution with techniques like salting keys or repartitioning.
- **Advanced DataFrame Operations**:
    - Use Window functions for complex analytics (e.g., ranking, running totals) with over() clause.
    - Implement custom UDFs (User-Defined Functions) for specialized data transformations.
- **Dynamic Resource Allocation**:
    - Enable spark.dynamicAllocation.enabled to scale executors based on workload, optimizing resource usage.
    - Configure spark.executor.memory and spark.executor.cores for efficient task execution.
- **Fault Tolerance Enhancements**:
    - Implement checkpointing with spark.checkpoint() to truncate RDD lineage for long-running jobs.
    - Use try-catch blocks in Spark applications to handle executor failures gracefully.
- **Integration with External Systems**:
    - Connect to streaming sources like Kafka using spark.readStream for real-time processing.
    - Write custom connectors for niche databases using Spark's DataSource API.

10. Spark Streaming (15 marks)

- **Definition**: Spark Streaming is a Spark module for processing real-time data streams, built on top of Spark Core, using a micro-batch processing model.
- **Core Concept**:
    - Processes data in small time intervals (micro-batches) as Discretized Streams (DStreams), which are sequences of RDDs.
    - Integrates seamlessly with Spark's batch processing for unified data pipelines.
- **Key Features**:
    - **Scalability**: Handles high-throughput streams by leveraging Spark's distributed architecture.
    - **Fault Tolerance**: Recovers from failures using RDD lineage and checkpointing.
    - **Integration**: Supports data sources like Kafka, Flume, Kinesis, and TCP sockets.
- **Programming Model**:
    - **DStream API**: Core abstraction for streaming, supporting transformations (e.g., map, filter) and actions (e.g., foreachRDD).

    - **Structured Streaming**: A higher-level API (since Spark 2.0) using DataFrames/Datasets for stream processing with SQL-like operations.
- **Data Sources**:
    - Kafka: Read streams using spark.readStream.format("kafka").
    - File Systems: Monitor directories for new files with spark.readStream.textFile().
    - Sockets: Process data from TCP connections for testing.
- **Output Sinks**:
    - Write stream results to consoles, files, databases, or Kafka using writeStream.
    - Supports modes like append, complete, or update for output handling.
- **Windowing and Aggregation**:
    - Perform time-based aggregations (e.g., count events per minute) using window operations on DStreams or Structured Streaming.
    - Example: df.groupBy(window("timestamp", "10 minutes")).count().
- **Checkpointing**:
    - Saves streaming state to fault-tolerant storage (e.g., HDFS) to recover from failures.
    - Enable with streamingContext.checkpoint(path) or writeStream.checkpointLocation.
- **Use Cases**:
    - Real-time analytics (e.g., monitoring website traffic).
    - Processing IoT sensor data or log streams.
    - Fraud detection with continuous data updates.

11. Machine Learning with Spark MLlib (15 marks)

- **Definition**: MLlib is Apache Spark's scalable machine learning library, designed for distributed data processing and integration with Spark's ecosystem.
- **Key Features**:
    - Scalable algorithms for large datasets, leveraging Spark's in-memory computing.
    - Supports classification, regression, clustering, recommendation, and more.
    - Integrates with DataFrames for seamless data preprocessing and model training.
- **Core Components**:
    - **Algorithms**:
        - Classification: Logistic Regression, Decision Trees, Random Forests, SVM.
        - Regression: Linear Regression, Generalized Linear Regression.
        - Clustering: K-Means, Gaussian Mixture Models.
        - Collaborative Filtering: Alternating Least Squares (ALS) for recommendations.
    - **Feature Engineering**:
        - Tools like VectorAssembler, StandardScaler, and StringIndexer for data preparation.

        - Supports TF-IDF, Word2Vec, and PCA for text and dimensionality reduction.
    - **Pipelines**:
        - Combines data preprocessing, feature extraction, and model training into a single workflow using Pipeline API.
        - Ensures reproducibility and modularity in ML workflows.
- **Model Evaluation**:
    - Provides evaluators like BinaryClassificationEvaluator and RegressionEvaluator.
    - Supports cross-validation and train-test splits via CrossValidator and TrainValidationSplit.
- **Distributed Training**:
    - Leverages Spark's distributed architecture to parallelize computations across clusters.
    - Handles large-scale datasets efficiently with fault tolerance.
- **Use Cases**:
    - Fraud detection (classification of suspicious transactions).
    - Customer segmentation (clustering for marketing).
    - Recommendation systems (e.g., product suggestions using ALS).
- **Programming Approach**:
    - Use DataFrame-based API (preferred) for structured data and pipeline workflows.
    - Example: Pipeline(stages=[VectorAssembler(), LogisticRegression()]).fit(df).
- **Integration**:
    - Combines with Spark SQL for data preprocessing and Spark Streaming for real-time predictions.
    - Exports models to PMML or saves them to HDFS for deployment.