1. How can Apache Spark fit into a data application? Include specific Spark functionalities that can be applied in a data application.

Apache Spark offers “lightning-fast cluster computing” for processing and analysing large datasets, using a core set of functionality complemented by a set of higher-level libraries like SparkSQL, Spark Streaming and MLlib.

Spark can connect to a variety of different storage systems like HDFS, Apache Cassandra, Apache HBase and more to allow seamless integration with data stores.

Spark can integrate with other big data technologies like Hadoop, Hive and Kafka for various data processing needs not met by the core Apache Spark functionality.

Spark can work with a range of higher-level engines that mean Spark has turned into a multifunctional data analytics tool designed for big data applications, rather than a simply being an engine for large-scale parallel and distributed data processing. Examples include Spark SQL for querying structured data via SQL or Hive, GraphX for graph analysis and graph-parallel computation, Spark Streaming for processing real-time streaming data and MLlib for machine learning.

1. Why is parallelism important and how does Spark parallelise tasks? Provide at least two specific examples for each.

Parallelism is a critical concept in distributed computing and big data processing, and it's essential for improving the performance, efficiency, and scalability of data processing tasks. In Apache Spark, parallelism is achieved through various mechanisms, and it plays a crucial role in maximising resource utilisation and reducing the time it takes to process large datasets.

How Spark Parallelises Tasks:

1. Data Parallelism:

Example 1: Distributed Data Processing: Spark splits large datasets into smaller partitions, and each partition is processed independently on different worker nodes. For instance, when performing a map operation on an RDD (Resilient Distributed Dataset), each partition can be processed in parallel.

Example 2: Parallel Data Transformations: When using Spark's DataFrame or Dataset APIs, operations like filtering, aggregation, and sorting can be parallelised across partitions, allowing multiple partitions to be processed simultaneously.

1. Task Parallelism:

Example 1: Parallel Execution of Stages: Spark divides a computation into stages, and tasks within a stage can be executed in parallel as long as they don't have dependencies on each other. For example, in a Spark application, a map stage and a reduce stage can run concurrently if there are no dependencies between them.

Example 2: Concurrent Execution of Jobs: Multiple Spark jobs can be submitted to a cluster, and they can run in parallel as long as they don't contend for the same resources. This enables concurrent execution of different data processing tasks.

1. Pipeline Parallelism:

Example 1: Chained Transformations: When you chain multiple Spark transformations together, Spark tries to optimise the execution by pipelining the tasks, so the data flows through the transformations without materialising intermediate results to disk. This reduces the need for expensive data shuffling and speeds up processing.

Example 2: Data Pipelines: In real-time streaming applications, Spark can parallelise data processing by constructing data pipelines with stages that perform different operations on the data as it flows through the pipeline.

1. What is a DataFrame in Spark and how is it different from a SQL table? Provide at least two specific examples for each.

A DataFrame in Spark is a distributed collection of data organised into named columns, similar to a table in a relational database or a spreadsheet in Excel. It is a fundamental data structure in Spark's higher-level API, designed to work with structured and semi-structured data. While DataFrames share similarities with SQL tables, they have some key differences.

DataFrame in Spark

* Structured Data Handling: DataFrames are designed for structured data processing. They store data in a tabular format with rows and columns, where each column has a name and a defined data type. This makes it suitable for handling data with schema information.

Example 1: Suppose you have a CSV file with sales data, and you want to work with it in Spark. You can load the CSV file into a DataFrame, where each column corresponds to a specific attribute like "product\_id," "sales\_amount," and "sales\_date."

* Support for Complex Operations: DataFrames offer a rich set of APIs for data manipulation and transformation. You can perform various operations like filtering, aggregation, joins, and window functions on DataFrames.

Example 2: You have a DataFrame containing customer data. You can easily perform operations like calculating the total spending per customer, filtering customers who made purchases above a certain threshold, and joining the customer data with another DataFrame containing order information.

SQL Table

Query Language: SQL tables are typically associated with a relational database management system (RDBMS) and are queried using SQL (Structured Query Language). SQL provides a standardised way to interact with and manipulate data in tables.

* Example 1: In an SQL database, you can create a table called "Employees" with columns like "EmployeeID," "FirstName," "LastName," and "Salary." You can then query this table using SQL statements, such as SELECT \* FROM Employees WHERE Salary > 50000.

Storage Location: SQL tables are stored in a database management system, which manages data persistence and ensures data integrity. SQL databases often have features like indexing, transactions, and foreign key constraints.

* Example 2: In a SQL database, you can define relationships between tables using foreign keys. For instance, you can create a "Customers" table and an "Orders" table, with a foreign key relationship between them, ensuring referential integrity when inserting or updating data.