**SPARK CLUSTER ASSIGNMENT**

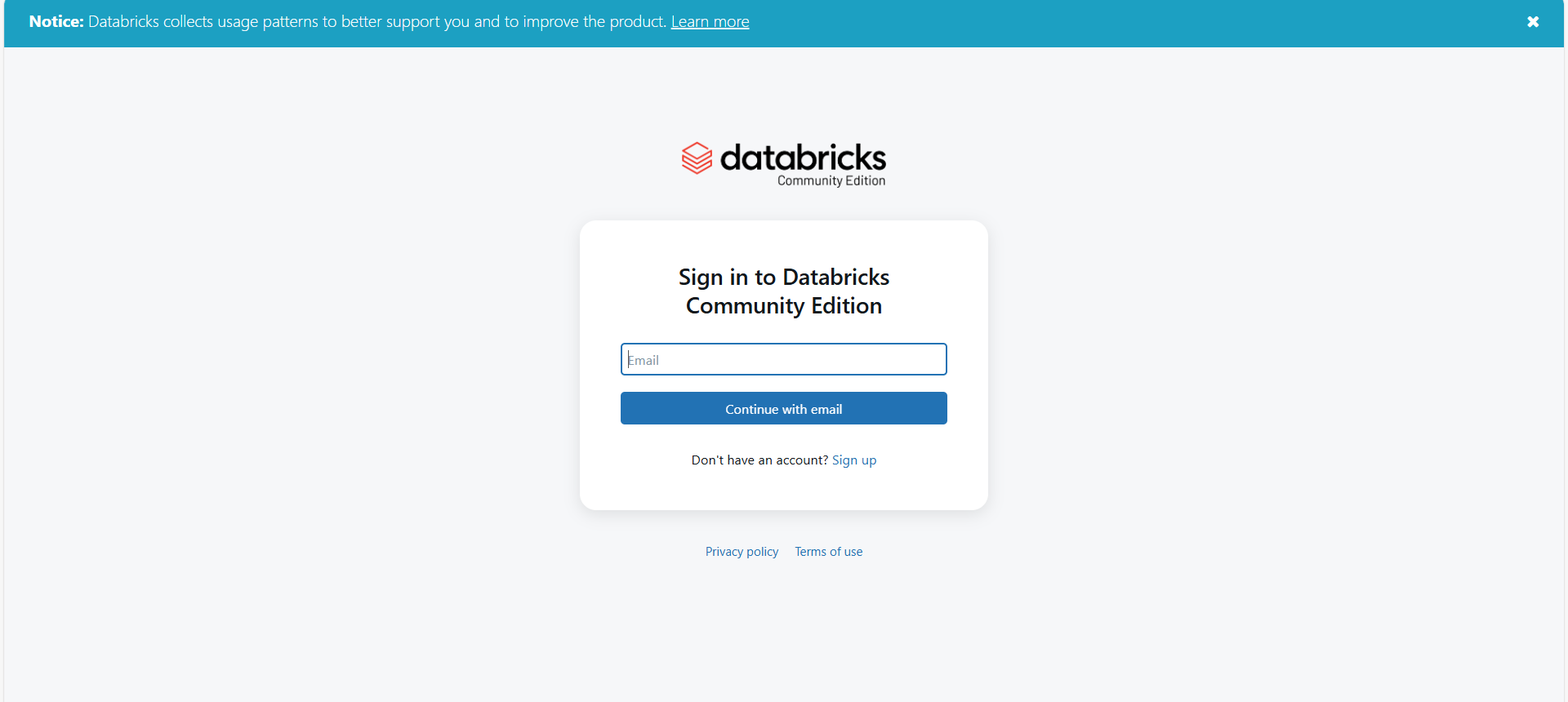
**Name:Tejaswini Gokanakonda**

## **Roll no**:**DE142**

## **Date:18-11-2024**

**Steps to Create a Cluster in Databricks**

1. **Log in to Databricks**
   * Navigate to [Databricks](https://databricks.com/) and log in to your account using your credentials.



1. **Go to the Workspace**
   * Once logged in, access the **Workspace** from the left-hand navigation bar.

**A screenshot of a computer

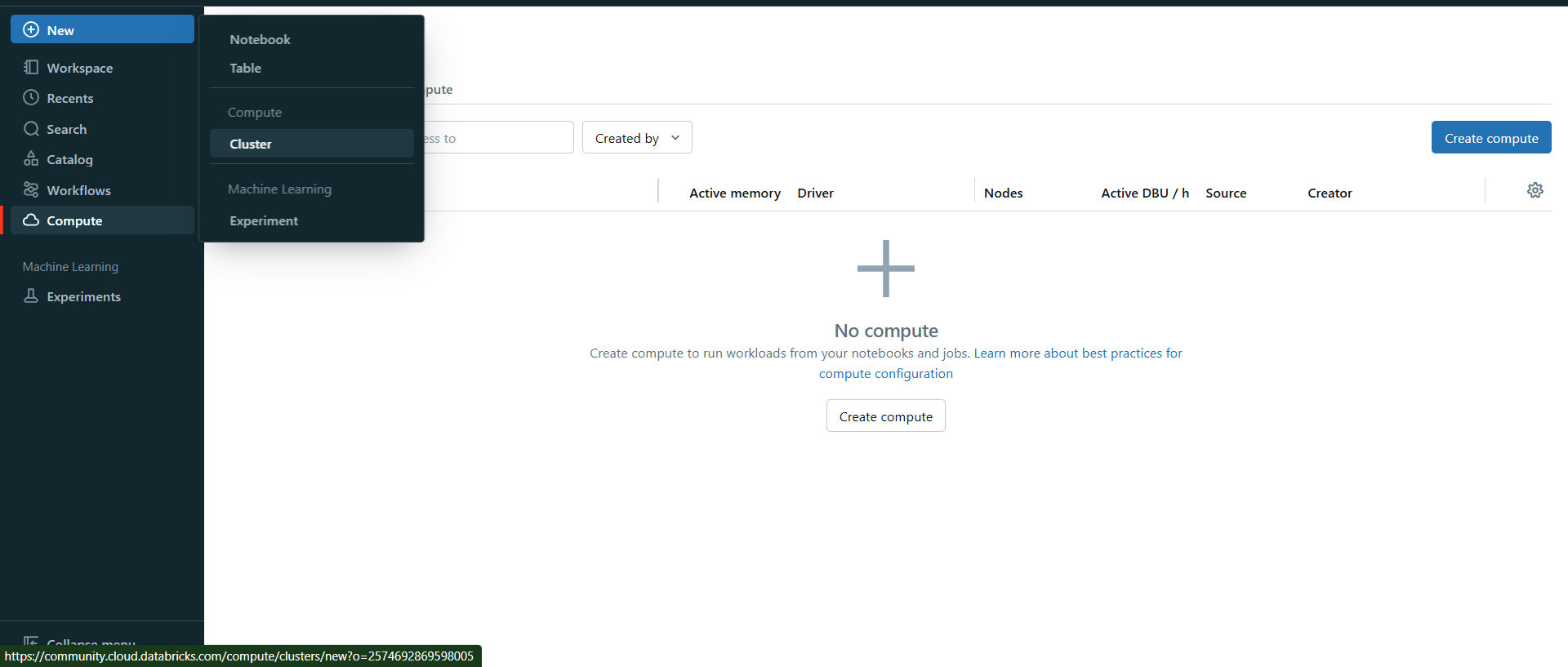
Description automatically generated**

1. **Open the Clusters Page**
   * Click on **Clusters** in the left-hand sidebar. This will open the Clusters page where you can create and manage clusters.

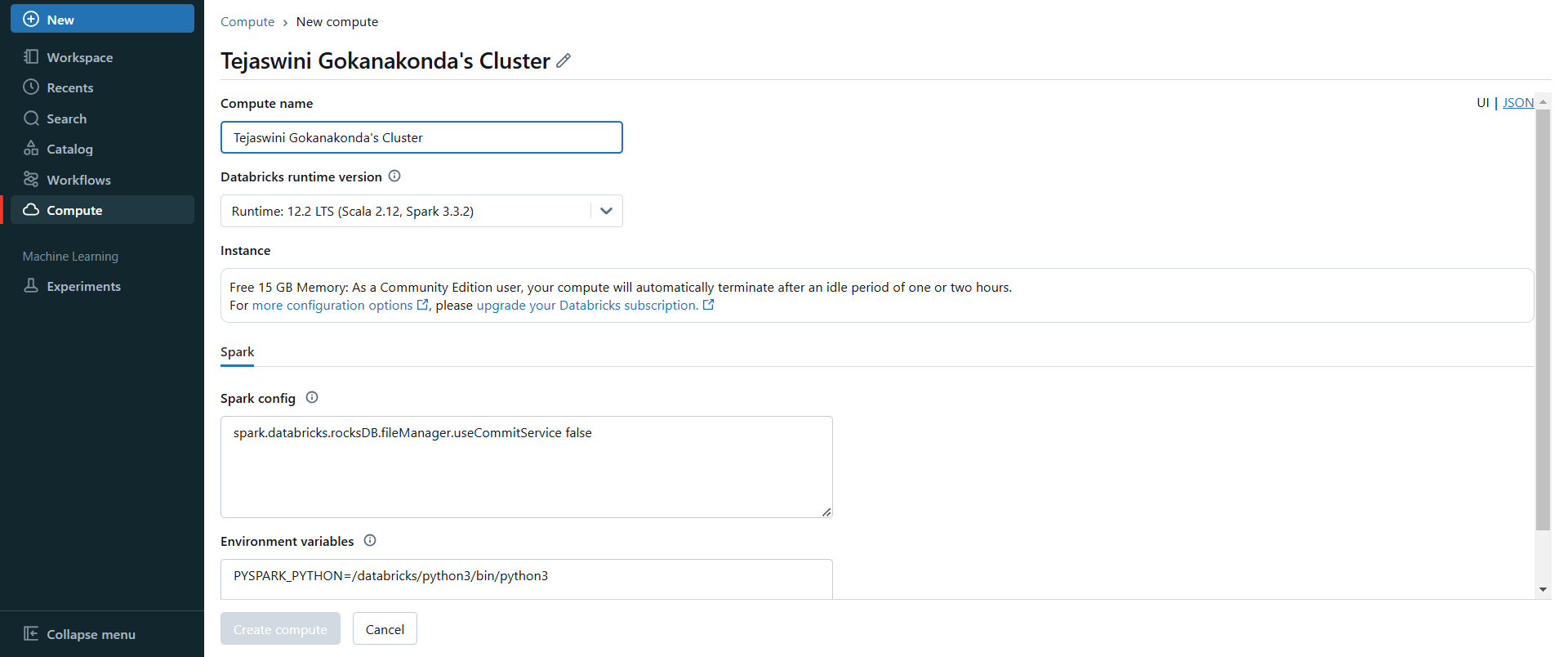
A screenshot of a computer

Description automatically generated

1. **Create a New Cluster**
   * Click on the **Create Cluster** button at the top of the Clusters page.

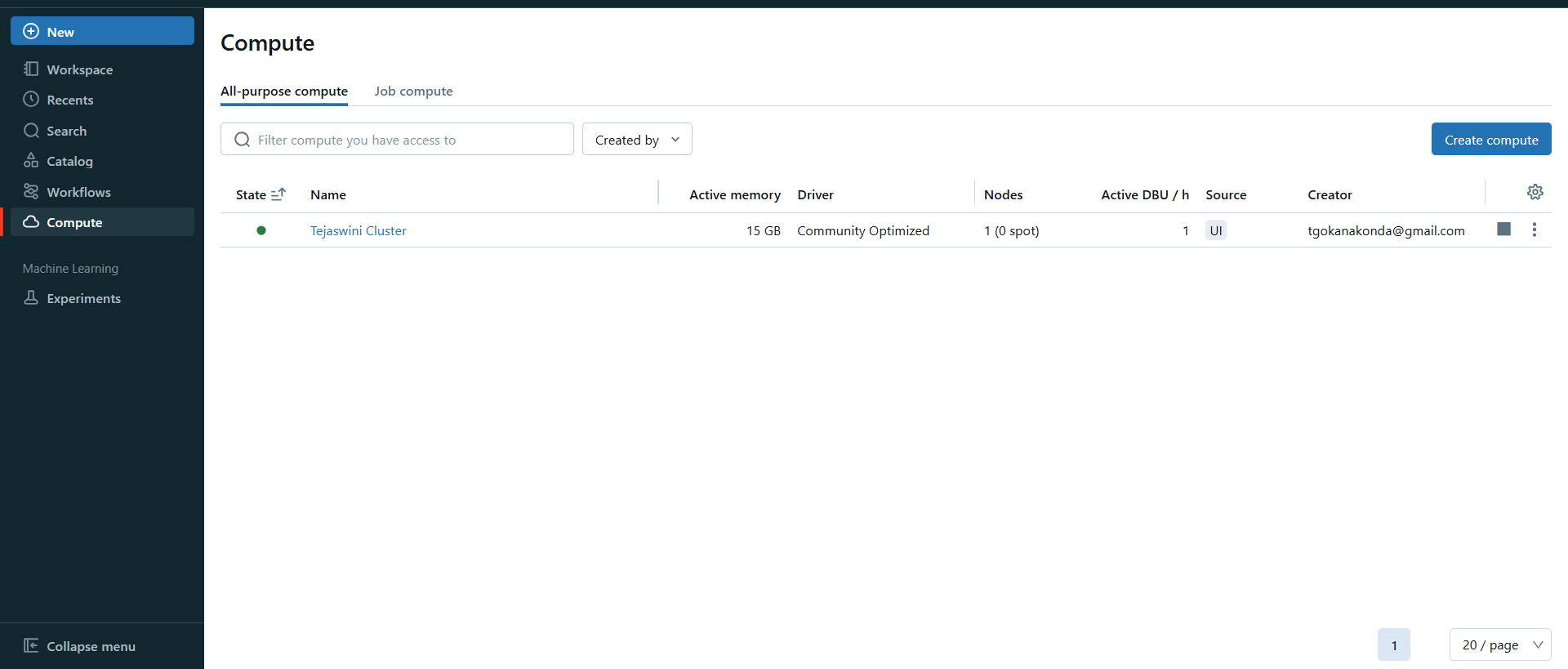


1. **Configure the Cluster Settings**
   * **Cluster Name**: Enter a name for your cluster.

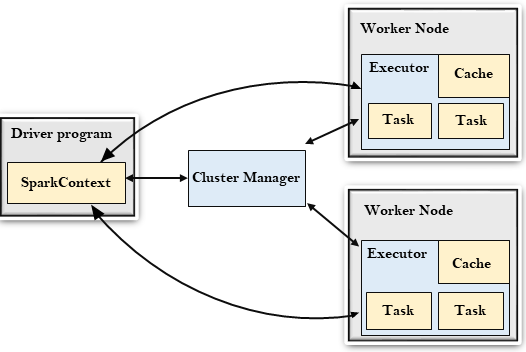


1. **Cluster Created**

Therefore the cluster is created Successfully



**Spark Architecture**



Apache Spark follows a master-slave architecture with two main components:

1. **Driver**: It coordinates the execution of tasks, maintaining the state of the application and the execution flow.
   * **SparkContext**: Acts as the entry point to Spark and handles the communication with the cluster manager.
   * **JobScheduler**: Schedules the jobs, splits them into stages, and coordinates execution.
2. **Cluster Manager**: Manages resources across the cluster. It can be:
   * **Standalone**: Spark’s own cluster manager.
   * **YARN**: Hadoop’s resource manager.
   * **Mesos**: A general-purpose cluster manager.
   * **Kubernetes**: Manages containerized applications.
3. **Workers**: Execute tasks assigned by the Driver. Each worker node runs an **Executor**, which performs the computation.

### Spark RDD (Resilient Distributed Dataset)

**RDD** is the fundamental data structure in Apache Spark, representing a collection of objects that can be processed in parallel across a cluster. It offers fault tolerance, meaning if any partition is lost, it can be recomputed using the lineage of transformations that created it. RDDs can be created from existing data or from operations on other RDDs.



The image depicts the architecture of Apache Spark’s Resilient Distributed Dataset (RDD), highlighting its capacity to handle distributed data processing efficiently. It showcases how data is ingested from storage, distributed across multiple nodes in memory, and processed concurrently. This architecture is designed to optimize performance by minimizing data movement and enabling parallel computation, which is essential for processing large datasets in a scalable and fault-tolerant manner. Spark’s ability to handle complex data transformations in a distributed environment is central to its high-performance capabilities.

**Components of the Architecture:**

* **"Data on Disk"**: This represents the raw data stored in a persistent storage system, typically visualized as a cylinder symbol, which is commonly used to depict databases or storage repositories.
* **"HDFS Read"**: Indicates that the data is being read from the Hadoop Distributed File System (HDFS), a popular distributed storage system designed for large-scale data processing.
* **"One Time Processing"**: Suggests that this is a batch processing operation, where the data is processed in a single pass, often used for large-scale data transformations.
* **"Distributed Memory"**: Depicted as a darker rectangular block, this represents the data being loaded into the memory of multiple nodes in a distributed system. It enables Spark to perform computations on the data without excessive reliance on disk storage.
* **Parallel Query Execution**: Multiple queries (**Query1, Query2, Query3**) are executed simultaneously on the same dataset, leveraging Spark’s ability to process data in parallel across the cluster.
* Each query generates distinct results (**Result1, Result2, Result3**), ensuring that computations are independent and efficiently distributed.

**Data Flow and Execution:**

* **Data Ingestion**: Initially, the raw data is loaded from persistent storage (e.g., HDFS), marking the starting point for processing.
* **Distributed Memory Allocation**: Once the data is read, it is distributed across the memory of several nodes, enabling parallel computation and reducing disk dependency.
* **Efficient Query Handling**: Multiple queries or transformations can be executed on the same dataset without needing to reload the data, minimizing unnecessary I/O operations.
* **Enhanced Performance**: This setup significantly boosts performance by allowing concurrent query execution and reducing data read and write cycles, ensuring faster data processing and optimized resource usage.

#### **Key Properties of RDDs:**

1. **Immutable**: Once created, an RDD cannot be changed. However, transformations can be applied to create new RDDs.
2. **Distributed**: Data in an RDD is partitioned across multiple nodes in the cluster, enabling parallel computation.
3. **Fault-tolerant**: RDDs keep track of the operations that created them in the form of a lineage, allowing lost data to be recomputed.
4. **Lazy Evaluation**: RDDs use lazy evaluation, meaning operations (transformations) are not computed until an action is performed.

#### **Operations on RDDs:**

1. **Transformations**: These create a new RDD from an existing one and are lazily evaluated.
   * **map(func)**: Applies a function to each element of the RDD and returns a new RDD.
   * **filter(func)**: Returns a new RDD containing only elements that satisfy the function.
   * **flatMap(func)**: Similar to map, but it can return 0 or more output elements for each input element.
   * **union(otherRDD)**: Returns an RDD that contains all elements from both RDDs.
   * **distinct()**: Removes duplicate elements in the RDD.
   * **groupByKey()**: Groups values by key (only for Pair RDDs).
2. **Actions**: These trigger computation and return a result.
   * **collect()**: Returns all the elements of the RDD to the driver as an array.
   * **reduce(func)**: Reduces the elements of the RDD using the given binary operator (e.g., sum, max).
   * **count()**: Returns the number of elements in the RDD.
   * **first()**: Returns the first element of the RDD.
   * **saveAsTextFile(path)**: Saves the RDD to a text file in HDFS, S3, or local storage.