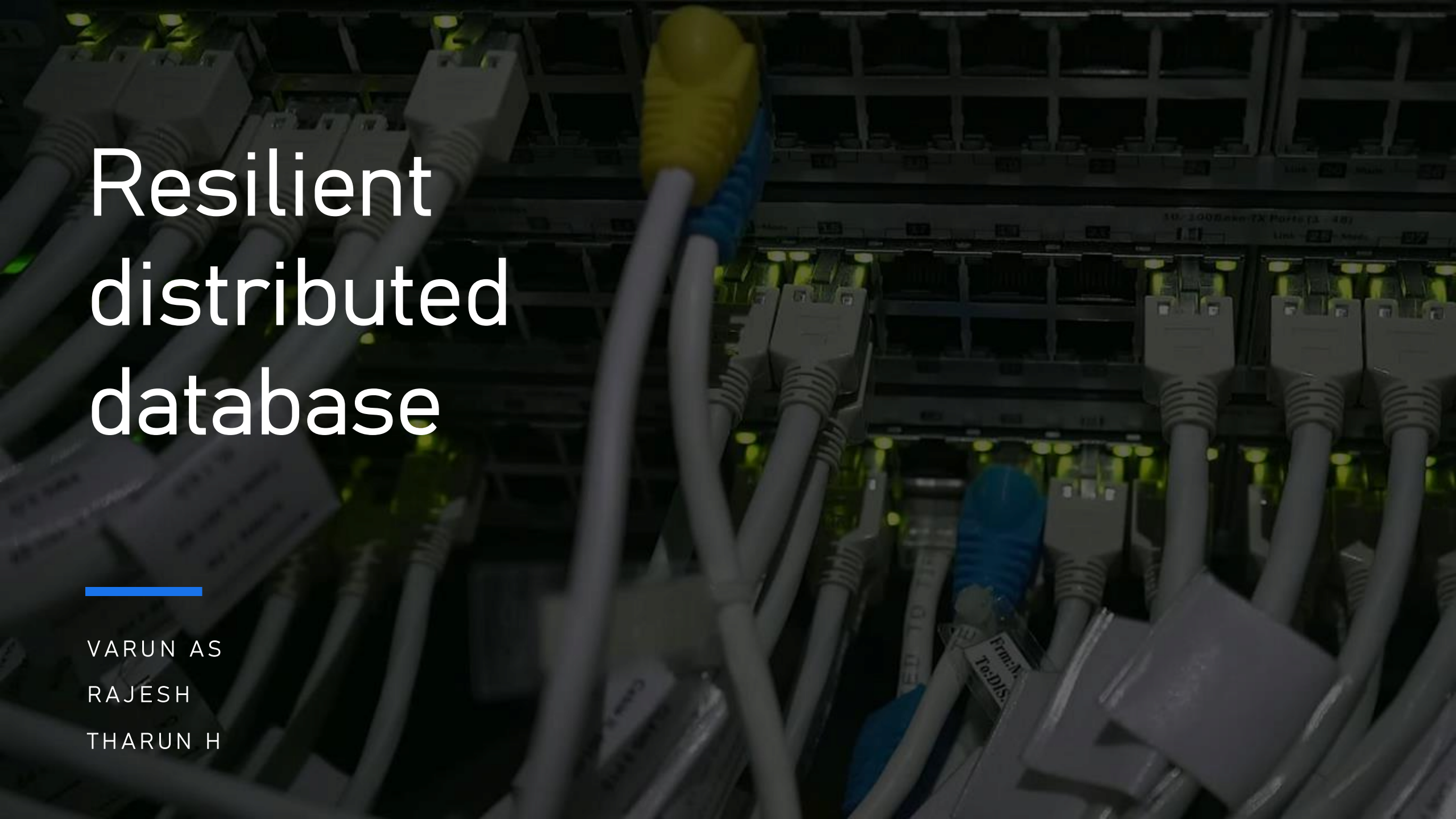


Resilient distributed database

VARUN AS
RAJESH
THARUN H

WHAT IS RDD

RDD stands for Resilient Distributed Dataset. RDDs are fundamental abstractions in Spark that represent a collection of elements partitioned across the nodes of a cluster, which can be operated on in parallel. They are resilient because they can automatically recover from failures, distributed because they can handle data across multiple nodes, and dataset because they represent data that can be processed in parallel.

RDDs in Spark are immutable, meaning once created, they cannot be changed. Instead, transformations applied to RDDs create new RDDs, allowing for a lineage of operations that can be optimized and computed efficiently across a distributed environment.

MAP REDUCE LIMITATIONS

1. **Data Sharing**: MapReduce requires data to be written to disk between map and reduce phases, leading to higher latency and lower performance for iterative algorithms compared to RDDs, which can keep data in memory across operations.
2. **Batch Processing**: MapReduce is primarily designed for batch processing rather than interactive or real-time processing, unlike RDDs which allow for more flexible data processing patterns.
3. **Complexity**: MapReduce programming can be more complex, requiring explicit handling of intermediate data and job orchestration. RDDs in Spark offer higher-level abstractions and optimizations that reduce the complexity of distributed data processing.
4. **Performance Overhead**: MapReduce involves more disk I/O due to its design, which can lead to higher latency and slower performance compared to RDDs, especially for iterative algorithms or tasks with frequent data access.
5. **Fault Tolerance**: While both MapReduce and RDDs offer fault tolerance, RDDs in Spark provide finer-grained control over fault tolerance and recovery mechanisms compared to the more rigid fault tolerance mechanisms of MapReduce.
6. **Optimization**: RDDs in Spark provide more opportunities for in-memory processing and optimizations compared to MapReduce, which relies heavily on disk I/O for data processing.

ADVANTAGES OF RDD

Fault Tolerance: RDDs are fault-tolerant by nature. Spark automatically tracks the lineage of transformations applied to RDDs, allowing lost data partitions to be recomputed based on the transformations applied to the original data.

In-Memory Processing: RDDs can store data in memory across cluster nodes, which significantly speeds up iterative algorithms and interactive data mining tasks compared to traditional disk-based systems like MapReduce.

Lazy Evaluation: RDDs support lazy evaluation, meaning transformations on RDDs are not computed immediately. Spark optimizes the execution plan and computes transformations only when an action requires a result, reducing unnecessary computations.

Ease of Use: RDDs provide a simpler and more flexible programming interface compared to MapReduce. They can be manipulated using familiar functional programming paradigms (like map, reduce, filter) in languages such as Python, Scala, and Java.

Versatility: RDDs are versatile and can handle a wide range of data types and operations, making them suitable for diverse data processing tasks including batch processing, iterative algorithms, interactive queries, and real-time stream processing.

Compatibility: RDDs seamlessly integrate with higher-level abstractions in Spark like DataFrames and Datasets, allowing developers to choose the level of abstraction that best fits their use case without sacrificing performance or flexibility.

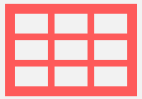
Community and Ecosystem: Spark's RDD model benefits from a vibrant community and extensive ecosystem of libraries and tools for data processing, machine learning, graph processing, and more, making it a robust choice for large-scale data analytics.

CORE CONCEPTS

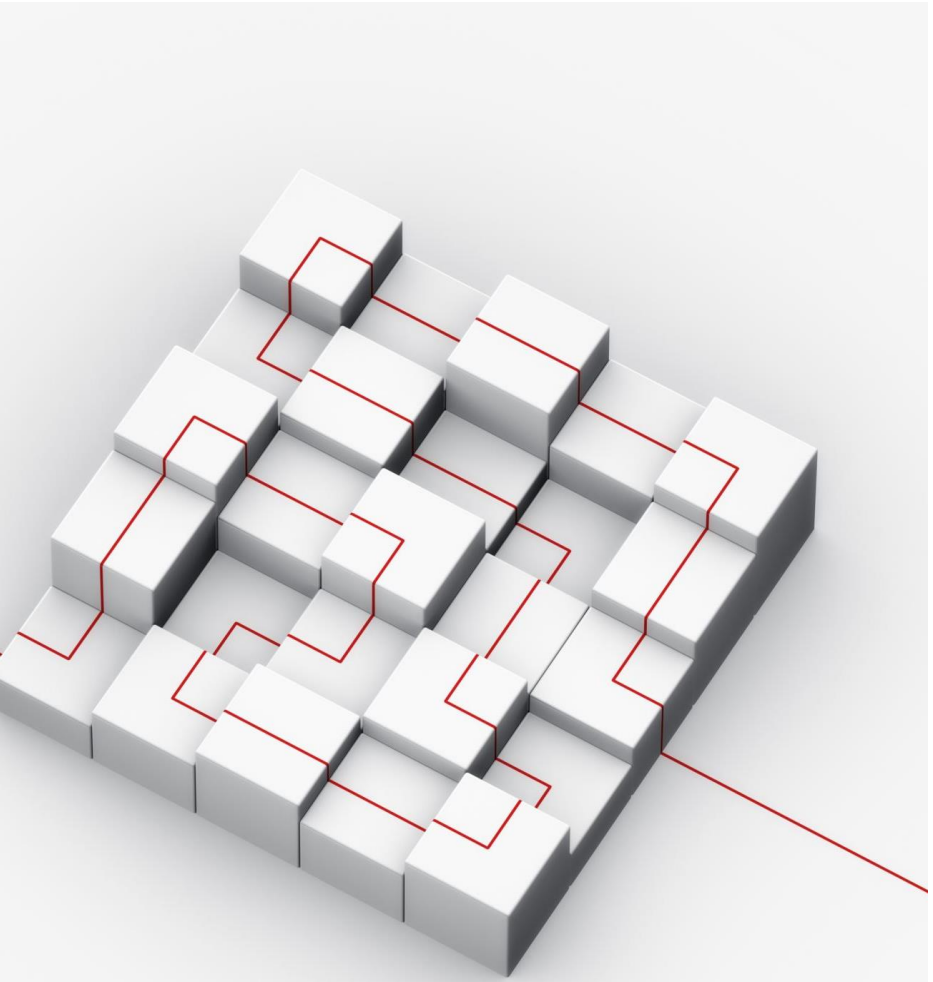
Abstraction: RDDs in Spark represent a distributed collection of objects that can be manipulated in parallel across a cluster. They are partitioned across nodes and can be operated on in parallel.

Immutability: RDDs are immutable, meaning once created, they cannot be modified. Instead, transformations applied to an RDD create new RDDs.

Lineage: RDDs maintain a lineage graph, which tracks the sequence of transformations applied to their base data. This lineage enables fault tolerance by allowing lost data partitions to be recomputed based on transformations.



Partitioning: RDDs are composed of one or more partitions, each of which is a logical division of data residing on a single machine in the cluster. Partitions are the units of parallelism and are processed in parallel across the cluster.



KEY METHODS

- **Creation**: RDDs can be created from stable storage like HDFS files, object stores (e.g., S3), existing Scala collections, or other RDDs through transformations.
- **Transformations**:
- **Map**: Applies a function to each element of the RDD.
- **Filter**: Selects elements that satisfy a predicate.
- **FlatMap**: Similar to `map`, but each input item can map to zero or more output items.
- **ReduceByKey**: Aggregates values for each key.
- **Join**: Performs an inner join between two RDDs based on their key.
- **Actions**:
- **Collect**: Retrieves all elements of the RDD as an array to the driver program.
- **Count**: Returns the number of elements in the RDD.
- **Reduce**: Aggregates the elements of the RDD using a function and returns a single result.
- **Persistence**: RDDs support persisting intermediate data in memory (MEMORY_ONLY), disk (DISK_ONLY), or both, which can improve performance for iterative algorithms and interactive data analysis.

RDD IMPLEMENTATION

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{Partition, SparkContext, TaskContext}

class CustomRDD(sc: SparkContext) extends RDD[String](sc, Nil) {

  override def compute(split: Partition, context: TaskContext): Iterator[String] = {
    // Logic to compute data for this partition
    val data = Seq("data1", "data2", "data3")
    data.iterator
  }

  override protected def getPartitions: Array[Partition] = {
    // Define partitions for the RDD
    Array(new CustomPartition)
  }
}

class CustomPartition extends Partition {
  // Custom partition implementation
  override def index: Int = 0
}
```

Integration with Spark Context

```
import org.apache.spark.{SparkConf, SparkContext}

object CustomRDDExample {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("Custom RDD Example").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Create a custom RDD
    val customRDD = new CustomRDD(sc)

    // Perform actions
    val result = customRDD.collect()
    result.foreach(println)

    sc.stop()
  }
}
```

REPRESENTING RDD

- Partition: RDDs are divided into logical partitions, which are basic units of parallelism in Spark. Each partition is a subset of the total data and resides on a single node in the cluster.
- Lineage: RDDs maintain lineage information, which tracks the sequence of transformations applied to the base dataset. This lineage enables fault tolerance by allowing lost partitions to be recomputed based on transformations.
- Operations: RDDs support two types of operations:
- Transformations: These are lazy operations that create a new RDD from an existing one (e.g., `map`, `filter`, `reduceByKey`). Transformations are not executed immediately but build a lineage graph.
- Actions: Actions trigger the execution of transformations to compute a result or perform an action (e.g., `collect`, `count`, `saveAsTextFile`).

RDD DEPENDENCIES

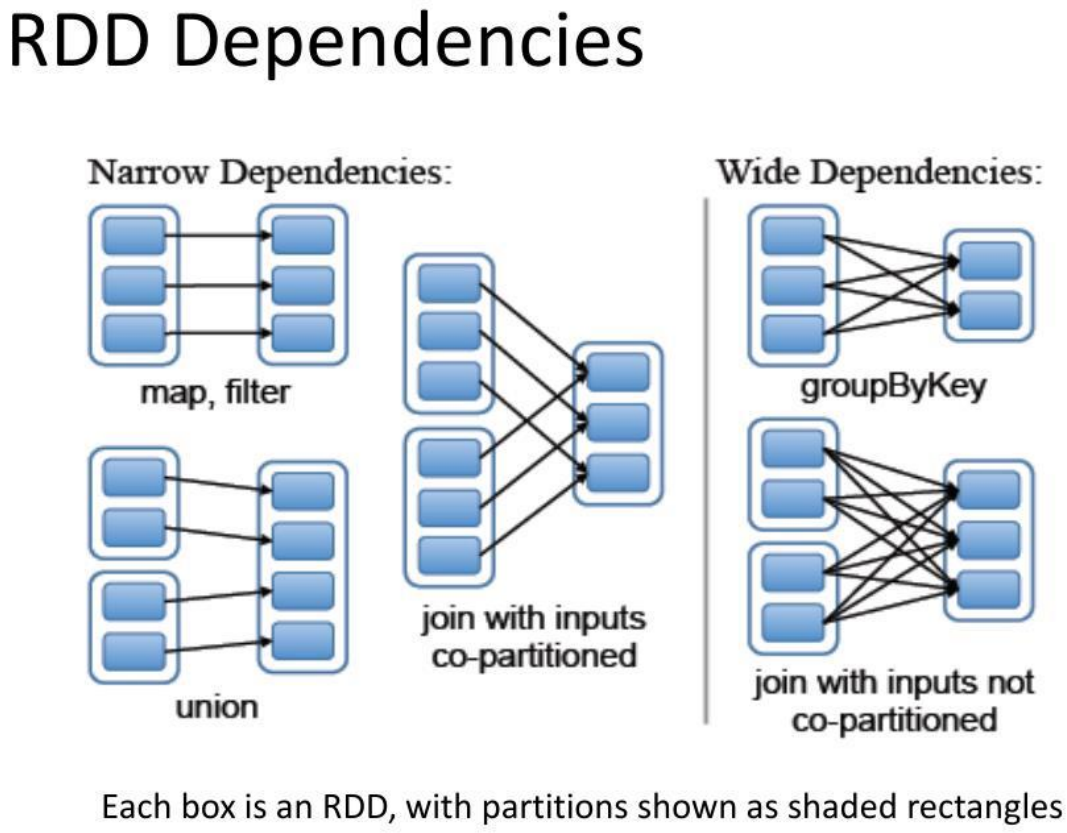
Narrow Dependencies (Narrow Transformations):

Definition: Narrow dependencies, also known as narrow transformations, occur when each partition of the parent RDD is used by at most one partition of the child RDD. This means that each output partition depends on a single input partition from the parent RDD, often in a one-to-one or one-to-few relationship.

Wide Dependencies (Wide Transformations):

Definition: Wide dependencies, or wide transformations, occur when each partition of the parent RDD is used by multiple partitions of the child RDD. This results in a one-to-many relationship between partitions of the parent and child RDDs.

RDD DEPENDENCIES



Each box is an RDD, with partitions shown as shaded rectangles

JOB SCHEDULING IN RDD

Action Trigger: When an action is called on an RDD, Spark initiates the job scheduling process. Actions are operations that trigger the actual execution of transformations defined on RDDs.

Stage Creation: Spark breaks down the execution plan (DAG) into stages. Each stage consists of tasks that can be executed in parallel, determined by the dependencies between RDDs (narrow or wide).

Task Scheduling: Spark's TaskScheduler assigns tasks to executor nodes based on data locality and available resources. It aims to minimize data movement (shuffle) and maximize parallelism.

Execution: Executors execute tasks in parallel across multiple nodes. Tasks within the same stage can run concurrently if their partitions are independent (narrow dependencies). For tasks with wide dependencies (e.g., shuffle operations like reduceByKey), Spark manages data exchange and synchronization across nodes.

Result Collection: Once all tasks are completed for a job, results are collected back to the driver program or stored in external storage systems.

MEMORY MANAGEMENT IN RDD

MEMORY_ONLY: Stores RDD partitions in memory as deserialized Java objects. This level provides fast access to data but can consume a significant amount of memory.

MEMORY_AND_DISK: If RDD partitions do not fit in memory, Spark spills them to disk and reads them back on demand. This level trades off memory usage for disk I/O operations.

MEMORY_ONLY_SER: Similar to MEMORY_ONLY but stores RDD partitions as serialized Java objects (binary data). This reduces memory usage but increases CPU overhead for serialization and deserialization.

MEMORY_AND_DISK_SER: Combination of MEMORY_AND_DISK and MEMORY_ONLY_SER, where RDD partitions are stored as serialized objects and spilled to disk if necessary.

DISK_ONLY: RDD partitions are stored only on disk, providing durability but slower access compared to memory-based storage levels.

CHECKPOINTING

- What is Checkpointing?
- Checkpointing involves saving the state of RDDs to reliable storage (such as HDFS, S3, or a distributed file system) to minimize the recomputation of RDDs in case of failures. It breaks the lineage chain of RDDs, making the current RDD state durable and reducing the dependency on the previous RDDs in the lineage.
- Purpose of Checkpointing:
- Fault Tolerance: Checkpointing provides fault tolerance by allowing Spark to recover RDD partitions from storage in case of executor failures or job restarts. It reduces the need to recompute RDDs from scratch.
- Lineage Control: Long lineage chains can consume significant memory and increase job recovery time. Checkpointing truncates these chains by replacing them with RDDs saved to storage, thereby improving performance for iterative algorithms.

HOW CHECKPOINTING WORKS

Checkpoint Directory: Spark requires a directory in a fault-tolerant file system (e.g., HDFS, S3) to store checkpoint data. The checkpoint directory should be accessible from all Spark executors.

Checkpoint Operation: To checkpoint an RDD, use the checkpoint() method on the RDD. This triggers the computation of the RDD lineage up to the checkpointed RDD and saves its partitions to the checkpoint directory.

Recovery: During job execution, if an RDD needs to be recomputed due to failure or eviction from memory, Spark retrieves its partitions from the checkpoint directory instead of recomputing from the source data or intermediate RDDs.

THANK YOU