

BIG DATA ANALYTICS

Parallel Programming with Spark

**Spark operations | Job execution |
Spark Applications**

Parallel Programming with Spark: *Outline*

- Overview of Spark
- Fundamentals of Scala and Functional Programming
- Spark Concepts
 - Resilient Distributed Datasets (RDD)
 - Creating RDDs
 - Basic Transformations
 - Basic Actions
 - Word Count Example
- **Spark operations**
- Job execution
- Spark Applications : Cluster computing with working sets



Spark operations

Transformations

- Transformations are operations that create a new RDD from an existing one. They are lazily evaluated.
- **Examples:**

```
// Read data from a text file
val lines = spark.textFile("hdfs://path/to/data.txt")
// Filter lines containing the word 'Spark'
val filtered = lines.filter(_.contains("Spark"))
// Split lines into words
val words = filtered.flatMap(_.split(" "))
// Map words to (word, 1) pairs
val wordPairs = words.map(word => (word, 1))
```

Spark operations


Actions

- **Actions** trigger the execution of transformations and return results to the driver or write them to storage.

- **Examples:**

```
// Count occurrences of each word
val wordCounts = wordPairs.reduceByKey(_ + _)
// Collect results to the driver
wordCounts.collect().foreach(println)
// Save results to HDFS
wordCounts.saveAsTextFile("hdfs://path/to/output")
```

Parallel Programming with Spark: *Outline*

- Overview of Spark
- Fundamentals of Scala and Functional Programming
- Spark Concepts
 - Resilient Distributed Datasets (RDD)
 - Creating RDDs
 - Basic Transformations
 - Basic Actions
 - Word Count Example
- Spark operations
- **Job execution** 
- Spark Applications : Cluster computing with working sets

Job execution

Job Execution Workflow

DAG (Directed Acyclic Graph):

- Transformations build a logical execution plan.
- Spark constructs a DAG for the workflow.

Stages and Tasks:

- A job is divided into stages, separated by shuffle boundaries.
- Each stage consists of multiple parallel tasks.

Lazy Evaluation:

- Transformations are evaluated only when an action is invoked.
- This allows Spark to optimize the execution plan.

Job execution

Example Workflow

```
// Read and process data

val data = spark.textFile("hdfs://path/to/data.txt")
val result = data.filter(_.contains("error"))
                  .map(line => (line.split(" ") (0), 1))
                  .reduceByKey(_ + _)

// Trigger execution and collect results
result.collect().foreach(println)
```

Job execution

Execution Flow

1. Transformations:

- `textFile`: Reads data.
- `filter`: Filters lines with "error".
- `map`: Maps lines to key-value pairs.
- `reduceByKey`: Aggregates values by key.

2. Actions:

`collect`: Triggers the execution and returns results.

Output:

- **Example:**

`(error1, 42)`

`(error2, 35)`

Parallel Programming with Spark: *Outline*

- Overview of Spark
- Fundamentals of Scala and Functional Programming
- Spark Concepts
 - Resilient Distributed Datasets (RDD)
 - Creating RDDs
 - Basic Transformations
 - Basic Actions
 - Word Count Example
- Spark operations
- Job execution
- **Spark Applications : Cluster computing with working sets**



Spark Applications : Cluster computing with working sets

Spark: Cluster Computing with Working Sets

- Leveraging Resilient Distributed Datasets (RDDs) for Iterative and Interactive Computing

Background:

- MapReduce limitations (acyclic data flow).
- Need for iterative machine learning algorithms and interactive analytics.

Purpose: Propose Spark for scalable, fault-tolerant cluster computing.

Spark Applications : Cluster computing with working sets

Spark Overview

- Leveraging Resilient Distributed Datasets (RDDs) for Iterative and Interactive Computing

Key Features:

- Fault tolerance
- Scalability
- Interactive capabilities

Core Concept: Resilient Distributed Datasets (RDDs)

Spark Applications : Cluster computing with working sets

Resilient Distributed Datasets (RDDs)

Definition:

- Read-only collections partitioned across machines.

Key Characteristics:

- Fault tolerance through lineage
- Lazy evaluation
- Memory caching for reuse

Spark Applications : Cluster computing with working sets

Programming Model

Driver Program:

- High-level control flow.

RDD Operations:

- Transformations (e.g., `map`, `filter`)
- Actions (e.g., `reduce`, `collect`)

Shared Variables:

- Broadcast variables
- Accumulators

Spark Applications : Cluster computing with working sets

Text Search Example

Objective:

- Count lines with "ERROR" in a log file.

Code Snippet:

```
val file = spark.textFile("hdfs://...")  
val errs = file.filter(_.contains("ERROR"))  
val ones = errs.map(_ => 1)  
val count = ones.reduce(_+_)
```

Output:

- Total error lines: 1234 (Example result based on input data).

Spark Applications : Cluster computing with working sets

Logistic Regression Example

Objective:

- Iterative classification using gradient descent.

Key Steps:

- Initialize random vector
- Update weights iteratively using cached data

Spark Applications : Cluster computing with working sets

Logistic Regression Example cont'd.

Code Snippet:

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D)
for (i <- 1 to ITERATIONS) {
  val grad = spark.accumulator(new Vector(D))
  for (p <- points) {
    val s = (1 / (1 + exp(-p.y * (w dot p.x))) - 1) * p.y
    grad += s * p.x
  }
  w -= grad.value
}
```

Output:

- Final weights vector: $[1.23, -0.45, 0.67]$ (Example result).
- Number of iterations: 10

Spark Applications : Cluster computing with working sets

Alternating Least Squares (ALS)

Objective: Collaborative filtering for recommendation systems.

Process:

- Optimize user and movie matrices iteratively.
- Use broadcast variables for efficiency.

Code Snippet:

```
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
  U = spark.parallelize(0 until u).map(j => updateUser(j, Rb, M)).collect()
  M = spark.parallelize(0 until m).map(j => updateUser(j, Rb, U)).collect()
}
```

Output:

- Final user matrix: `[[0.5, 0.3], [0.1, 0.7], ...]`
- Final movie matrix: `[[0.4, 0.2], [0.9, 0.6], ...]`

Spark Applications : Cluster computing with working sets

Implementation Insights

Architecture:

- Built on Mesos.
- RDD lineage ensures fault tolerance.

Task Scheduling:

- Data locality using delay scheduling.

Shared Variables:

- Efficient serialization and caching.

Spark Applications : Cluster computing with working sets

Experimental Results

Logistic Regression:

- Spark vs. Hadoop performance comparison.
- Faster iterations due to caching.
- **Example:** 10x speedup after initial iteration.

Alternating Least Squares (ALS):

- Improved performance with broadcast variables.
- **Example:** 2.8x speedup for 30-node cluster.

Spark Applications : Cluster computing with working sets

Discussion

Strengths:

- Handles iterative and interactive workloads.
- Fault tolerance with minimal overhead.

Limitations:

- Prototype stage.
- Dependency on Scala.

Spark Applications : Cluster computing with working sets

Conclusion

Summary:

- Spark introduces RDDs for efficient data reuse.
- Outperforms traditional systems for iterative jobs.

Future Enhancements:

- Shuffle operations for group-by and joins.
- Higher-level interfaces (e.g., SQL, R).