04 Shared Variables -KirkYagami 👨‍💻🕵️

Special variables that can be used in parallel operations. PySpark provides two types of shared variables:

03 Spark coding(RDD) | Spark Basics

1. Broadcast Variables in PySpark

Key Points:

- **Definition**: Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- **Purpose**: They are useful for sharing large datasets across all nodes to avoid the overhead of serializing and deserializing the dataset multiple times.
- **Read-Only**: Broadcast variables are read-only; they cannot be modified by the tasks once broadcasted.

```
from pyspark import SparkConf, SparkContext
import collections


conf = SparkConf().setMaster("local").setAppName("SharedVariables")
conf.set("spark.local.dir", "C:/Users/NikhilSharma/Desktop/spark_course/tmp")
sc = SparkContext(conf=conf)


#boardcating the variable
bc = sc.broadcast(["python", "databricks", "deltalake", "spark", "azure"])
#calling the boardcasted variable
var = bc.value # returns a list

print(type(var))
```

Practical Use Cases:

1. **Lookup Tables**:
   - **Scenario**: You have a large lookup table that needs to be used by each node to map some values in a large RDD.
   - **Example**:

```
from pyspark import SparkConf, SparkContext
import collections
```

```
conf = SparkConf().setMaster("local").setAppName("SharedVariables")
conf.set("spark.local.dir", "C:/Users/NikhilSharma/Desktop/spark_course/tmp")
sc = SparkContext(conf=conf)

# Large lookup table
lookup_table = {1: "one", 2: "two", 3: "three"}

# Broadcast the lookup table
broadcast_lookup = sc.broadcast(lookup_table)

# Data RDD
data = sc.parallelize([1, 2, 3, 4, 5])

# Use the broadcast variable in a transformation
result = data.map(lambda x: broadcast_lookup.value.get(x,
"unknown")).collect()

print(result)  # Output: ['one', 'two', 'three', 'unknown', 'unknown']
```

2. **Machine Learning Models**:
   - **Scenario**: You have a trained machine learning model that needs to be applied to a large dataset distributed across many nodes.
   - **Example**:

```
import joblib
from pyspark import SparkConf, SparkContext
import collections


conf = SparkConf().setMaster("local").setAppName("SharedVariables")
conf.set("spark.local.dir", "C:/Users/NikhilSharma/Desktop/spark_course/tmp")
sc = SparkContext(conf=conf)

sc = SparkContext("local", "Broadcast ML Model Example")

# Load a pre-trained model (assuming a scikit-learn model)
model = joblib.load('path/to/model.pkl')

# Broadcast the model
broadcast_model = sc.broadcast(model)

# Data RDD (features)
data = sc.parallelize([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
```

```
# Use the broadcast variable in a transformation to make predictions
result = data.map(lambda features: broadcast_model.value.predict([features])
[0]).collect()

print(result)  # Output: Predictions from the model
```

2. Accumulators in PySpark

Key Points:

- **Definition**: Accumulators are variables that are only "added" to through an associative and commutative operation and can be used to implement counters or sums.
- **Purpose**: They are useful for aggregating information across the cluster, such as counting events or accumulating values.
- **Write-Only from Workers**: Accumulators can be updated from the workers, but the value is only reliably read by the driver program.

Behavior and Characteristics:

- **Initialization**:
  - An accumulator is created on the driver and initialized to a given value.
- **Update Mechanism**:
  - Tasks running on workers add to the accumulator using operations like `+=`.
  - Each task has its own local copy of the accumulator to which it writes.
- **Aggregation**:
  - The driver program periodically aggregates these updates.
  - The aggregated value can be accessed on the driver program using the `value` attribute.

Practical Use Cases:

1. **Counting Events**:
   - **Scenario**: Counting the number of occurrences of a specific event (e.g., the number of erroneous records).
   - **Example**:

```
from pyspark import SparkConf, SparkContext
import collections


conf = SparkConf().setMaster("local").setAppName("SharedVariables")
conf.set("spark.local.dir", "C:/Users/NikhilSharma/Desktop/spark_course/tmp")
sc = SparkContext(conf=conf)

# Create an accumulator
error_count = sc.accumulator(0) # error count is
```

```
# Data RDD
data = sc.parallelize(["good", "bad", "good", "bad", "good"])

# Function to identify and count errors
def process_record(record):
    global error_count
    if record == "bad":
        error_count += 1
    return record

# Process data
result = data.map(process_record).collect()

# The value of the accumulator on the driver
print(f"Number of errors: {error_count.value}")  # Output: Number of errors:
2
```

2. **Summing Values**:
   - **Scenario**: Summing a specific metric across a large dataset (e.g., total sales amount).
   - **Example**:

```
from pyspark import SparkConf, SparkContext
import collections


conf = SparkConf().setMaster("local").setAppName("SharedVariables")
conf.set("spark.local.dir", "C:/Users/NikhilSharma/Desktop/spark_course/tmp")
sc = SparkContext(conf=conf)

V4842612
# Create an accumulator
total_sales = sc.accumulator(0.0)

# Data RDD
data = sc.parallelize([100.0, 200.5, 300.0, 150.75, 250.25])

# Function to add sales amount to the accumulator
def add_sales(sale):
    global total_sales
    total_sales += sale

# Apply the function
data.foreach(add_sales)

# The value of the accumulator on the driver
print(f"Total sales: {total_sales.value}")  # Output: Total sales: 1001.5
```

3. **Debugging and Logging**:
   - **Scenario**: Counting the number of specific types of log messages (e.g., the number of warnings or errors).
   - **Example**:

```
sc = SparkContext("local", "Accumulator Logging Example")

# Create accumulators for different log levels
warning_count = sc.accumulator(0)
error_count = sc.accumulator(0)

# Log messages RDD
logs = sc.parallelize(["INFO: All good", "WARN: Something might be wrong",
"ERROR: Something is wrong", "WARN: Check this out"])

# Function to count warnings and errors
def count_logs(log):
        global warning_count, error_count
        if "WARN" in log:
                warning_count += 1
        elif "ERROR" in log:
                error_count += 1

# Apply the function
logs.foreach(count_logs)

# The values of the accumulators on the driver
print(f"Number of warnings: {warning_count.value}")  # Output: Number of
warnings: 2
print(f"Number of errors: {error_count.value}")      # Output: Number of
errors: 1
```

Summary

Accumulators in PySpark are a powerful tool for aggregating information across a distributed cluster. They are useful for tasks such as counting events, summing values, debugging and logging, and progress tracking. By using accumulators, you can efficiently collect metrics and statistics from worker nodes to the driver node.