Understanding Task Distribution in Spark: Avoiding Closure Issues and Ensuring Accurate Result Aggregation

prabakar s

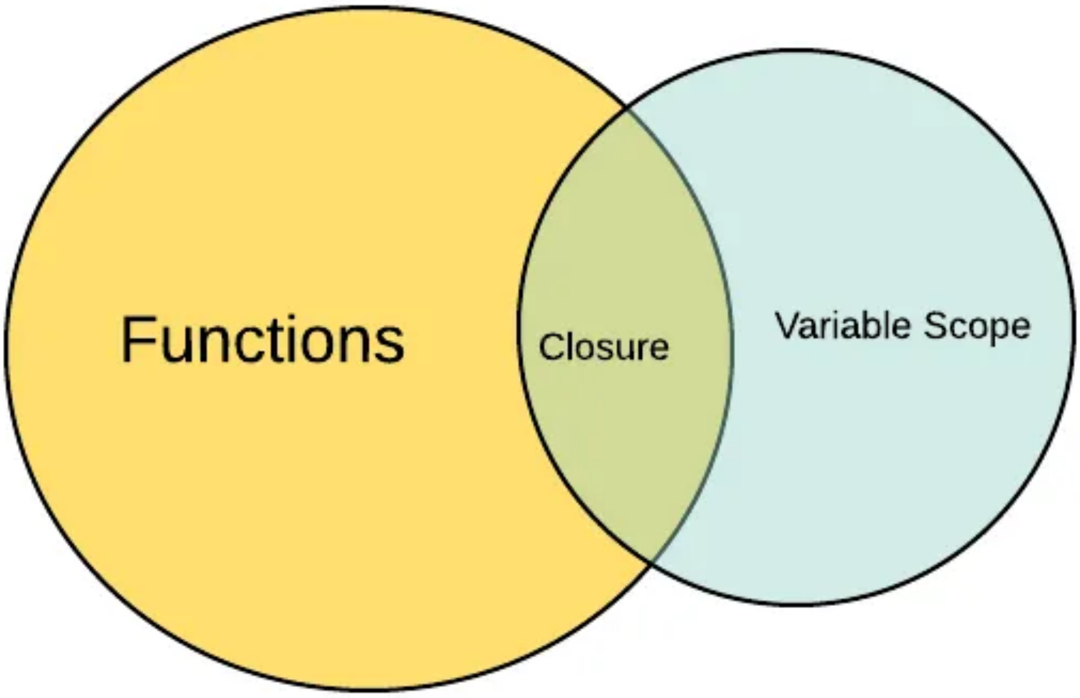
7 min read · Sep 18, 2024

Hello Data Folk, In this blog we are going to see about the closure in pyspark.

What is a closure in programming?

In general, a **closure** is a function or a block of code that "remembers" the environment in which it was created, even after that environment has ceased to exist.

Because Spark operates in a distributed manner, it cannot preserve the environment for closures



Understanding How Spark Program Works

> *Let’s understand how Spark handles variables and methods when running tasks*

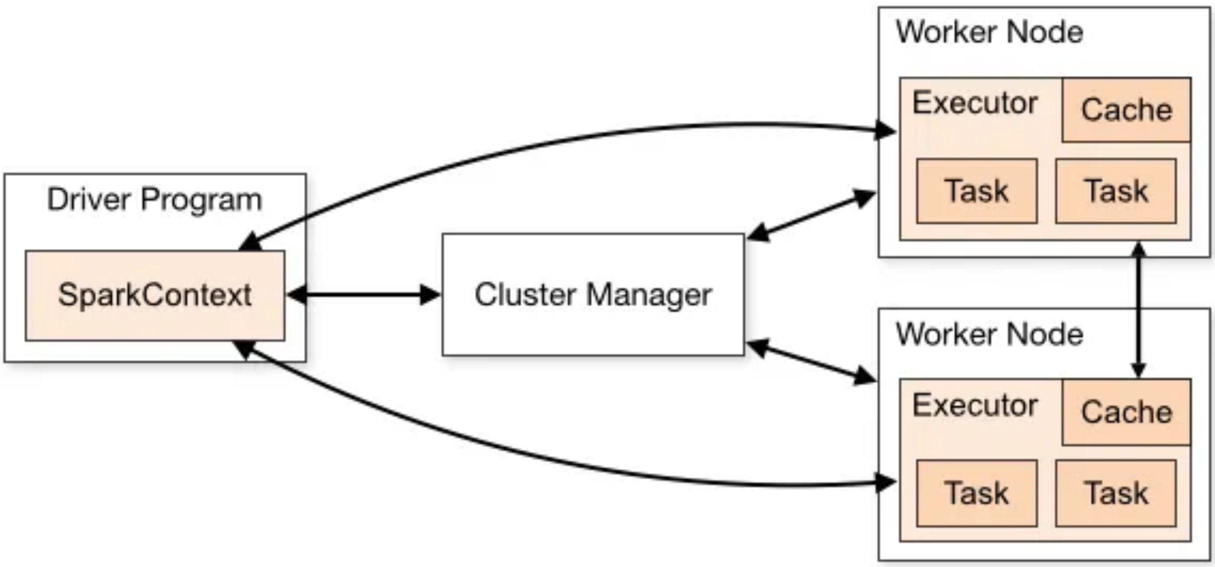
1. **Local & Cluster Mode:**

- **Local mode:** PySpark behaves similarly to how you’d run a normal Python script. It can still use parallelism by running multiple processes on your local machine.
- **Cluster mode:** This is where the code is executed across multiple machines (nodes) in a cluster. The challenge is that the Spark program runs differently in this setup, particularly in how it handles variables and methods.

We are going to focus on what happens on cluster mode when spark code is submitted. Let's assume we have a spark cluster with 2 nodes

2. Driver and Executors:

- In cluster mode, the **driver** node manages the execution of your Spark job. It creates a Directed Acyclic Graph (DAG) of tasks and distributes these tasks to worker nodes, called **executors**.
- **Executors** are the nodes that perform the actual computation on chunks of data. The driver sends them instructions on which tasks to run.



3. Closures:

- Before the executors can run their tasks, the driver prepares something called a **closure**. ***A closure is a set of variables and methods that the worker nodes will use during task execution.***
- Each worker gets its own **copy** of these variables and methods, which

is important because any changes that happen to these variables inside the executor will **not** reflect on the driver or other workers. This means variables and methods are isolated in each worker.

Key Issue (Global vs. Local Scope):

- If your code modifies a variable inside an executor (for example, a counter), that change is **local** to that executor. It won't affect the global state of the variable or the driver's copy.
- This can be confusing when you expect a global change (like increasing a counter globally) but instead, every worker operates on its own version of the variable, potentially leading to incorrect or unexpected results.
- This behavior is crucial to understand when moving from local to cluster mode because bugs can be hard to track down if you assume changes to a variable in one part of the cluster will be visible in another.

Let's go and understand this with an example,

```
total_count = 0
def process(row):
   global total_count
   total_count += row['value']
rdd.foreach(process)
```

Step-by-Step Execution in a Spark Job

1. Driver Program (Main Application):

The **driver** is the program that you write in PySpark. It handles the main control flow of the Spark application, submits jobs, and coordinates the execution.

In this example, the driver has:

- A global variable `total_count` initialized to `0`.
- A function `process()` which takes a row, accesses the global variable, and increments `total_count` by the value from the row.

2. RDD (Resilient Distributed Dataset):

The data in Spark is split across multiple partitions (think of partitions as chunks of the dataset). Let's assume the `rdd` is distributed across two partitions (because we have a two-node cluster), with each node responsible for one partition of the data.

Example:

- Partition 1: `[{'value': 3}, {'value': 4}]`
- Partition 2: `[{'value': 5}, {'value': 6}]`

3. Job Submission:

When you call `rdd.foreach(process)`, you are submitting a **job** to Spark.

The driver:

- Creates a **DAG (Directed Acyclic Graph)** of tasks.
- Decides to split the job into multiple **tasks**, based on how the `rdd` is partitioned (2 partitions → 2 tasks).

Driver actions:

- Each task corresponds to processing one partition of the RDD.
- The driver serializes the `process()` function and the `total_count` variable (captured in the **closure**) and sends them to the worker nodes (executors). Here serialization means converting functions into binary codes like pickling in python.

4. Task Execution on Executors:

The two executors (on two nodes) each get one task to process a partition of the RDD.

- **Executor 1** gets Partition 1 (`[{'value': 3}, {'value': 4}]`).
- **Executor 2** gets Partition 2 (`[{'value': 5}, {'value': 6}]`).

Each executor also receives the closure, which includes a copy of the `total_count` variable (initialized to 0) and the `process()` function.

> *Executor 1:*

- `Local variable total_count = 0`

- Processes the first row: `total_count += 3` → `total_count = 3`.
- Processes the second row: `total_count += 4` → `total_count = 7`.

> *Executor 2:*

- `Local variable total_count = 0`
- Processes the first row: `total_count += 5` → `total_count = 5`.
- Processes the second row: `total_count += 6` → `total_count = 11`.

Important: Each executor is working with its **own copy** of `total_count` in isolation. So, `total_count` inside **Executor 1** is completely separate from `total_count` inside **Executor 2**.

5. No Global Aggregation:

After both executors complete their tasks, the result is returned to the driver, but the global `total_count` on the driver **remains unchanged at 0**. This is because:

- The changes to `total_count` happened locally on each executor and were not communicated back to the driver.
- There's no automatic mechanism in place to aggregate or sum the `total_count` values from the executors.

Key Problem:

- **Global Scope Issue:** The `total_count` variable inside the executors is a local copy, and its updates are not reflected in the driver or across other nodes. Each executor is working independently with its own version of `total_count`, so there's no communication or aggregation of the total count between them.

How to Fix It?

To properly aggregate the results across all nodes, we need to use a **Spark action** that supports aggregation, like `reduce()` or `aggregate()`.

Fixed Example Using `reduce`:

```
rdd.map(lambda row: row['value']).reduce(lambda x, y: x + y)
```

Explanation of Fixed Code:

Step-by-Step Execution of the New Code

1. Driver Program (Main Application):

- The driver submits the job to process an RDD.
- This time, instead of relying on a global variable (`total_count`), the job is broken into smaller, distributed tasks using transformations (`map()`) and actions (`reduce()`).

2. RDD with Two Partitions:

Let's assume the RDD contains the following data, distributed across two partitions (because we are using a two-node cluster):

- **Partition 1:** `[{‘value’: 3}, {‘value’: 4}]`
- **Partition 2:** `[{‘value’: 5}, {‘value’: 6}]`

3. Transformation — `map()` :

The first step in the pipeline is the `map()` transformation:

```
rdd.map(lambda row: row['value'])
```

- **What `map()` Does:** This transformation is applied to each element (row) of the RDD. It extracts the `value` field from each row and creates a new RDD consisting of these extracted values.
- After applying `map()` :
- **Partition 1** becomes: `[3, 4]`
- **Partition 2** becomes: `[5, 6]`

Each partition is still distributed across two nodes (executors), and the `map()` function is applied in parallel on each partition locally. There's no issue with closure here because we aren’t relying on any global variables —each worker simply extracts values from the rows in its partition.

4. Action — `reduce()` :

The next step is applying the `reduce()` action:

```
reduce(lambda x, y: x + y)
```

- **What `reduce()` Does:** `reduce()` takes two elements at a time (in this case, the values in the RDD) and combines them according to the lambda function (in this case, summing them up: `x + y`).

> *Partition-Level Reduce:*

- Spark first performs the `reduce()` **locally** within each partition to aggregate the results. This is done independently for each partition:
- **Partition 1:** It reduces `[3, 4]` → `3 + 4 = 7`.
- **Partition 2:** It reduces `[5, 6]` → `5 + 6 = 11`.
- Now each partition has a single value:
- **Partition 1 Result:** `7`
- **Partition 2 Result:** `11`

5. Final Aggregation Across Partitions:

After the local reduction within each partition, Spark performs a **global reduction** to aggregate the results from all partitions.

- Spark takes the partial results from each partition:
- Result from Partition 1: `7`

- It then applies the `reduce()` operation one final time at the driver level: `7 + 11 = 18`

This final reduction happens in memory on the driver node, which aggregates the results from all the partitions into a single final result.

6. Final Result:

The driver now has the final result, `18`, which is the sum of all `value` fields across all partitions.

Why This Fix Works:

- **No Global Variables**: There is no reliance on global variables like `total_count`. Instead, the summing operation is fully handled by Spark's distributed mechanism (using `reduce()`).
- **Distributed Aggregation**: The `reduce()` function operates first locally (within each partition) and then globally (across partitions). This ensures that the result is correctly aggregated from all partitions.
- **Parallel Processing:** The `map()` and `reduce()` transformations allow Spark to process the data in parallel across multiple nodes. Each node handles its partition independently, and Spark automatically takes care of combining the results.

Comparison with the Original Code:

- In the original code, each executor had its own isolated copy of `total_count`, and changes were not reflected globally. This led to the **closure issue**.

- In the fixed code, Spark handles all aggregations using transformations and actions that are **designed for parallel processing.** There's no need for shared state or global variables, so there's no closure issue.

Conclusion:

In this blog, we've explored how the Spark driver distributes tasks across executors, leading to potential issues with local and global scope. We also discussed how this problem arises due to variable handling in a distributed environment, and how we can resolve it by modifying the code to ensure proper parallel processing and result aggregation.

I hope you find this content informative and useful for your learning. Thank you for reading, and happy data engineering! :)

Pyspark Data Engineering Spark Pyspark Closure

Written by prabakar s

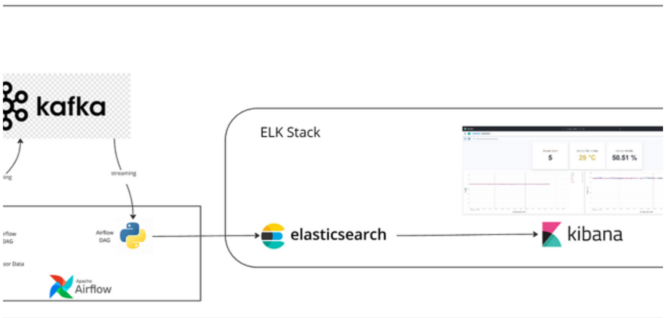
18 followers · 14 following

No responses yet

A Aaditya Adhikari

What are your thoughts?

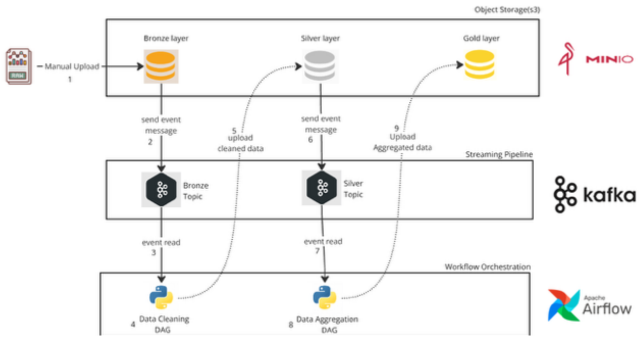
More from prabakar s



prabakar s

Building Data Pipeline Project with Apache Kafka, Airflow &...

Sep 2, 2024 301



prabakar s

Medallion Architecture: Building Multi-Hop Data Processing usin...

In this blog, we'll explore how to implement the Medallion Architecture with a practical...

Sep 6, 2024 308 1

See all from prabakar s

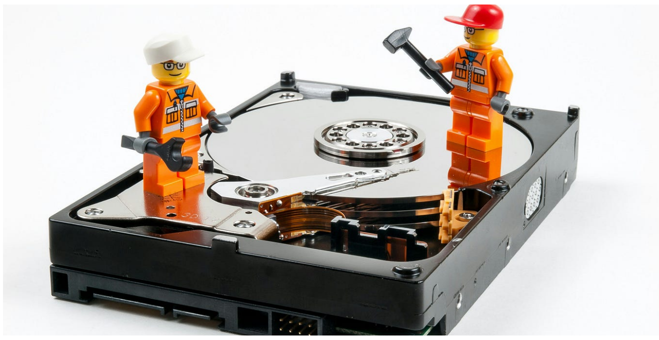
Recommended from Medium

A Mishra Ankur

PySpark Optimization: Best Practices for Better Performance

PySpark Optimization: Best Practices for Better Performance

Mar 3 2



In Data Engineer Things by Santosh Joshi

Caching and Persisting in PySpark

Explore Differences Between Caching and Persisting in PySpark

Dec 30, 2024 17

Priyanshu Rajput

Kafka Showdown: poll() vs consume() 🚀—Which One...

When working with Apache Kafka, you'll often need to retrieve messages from a...

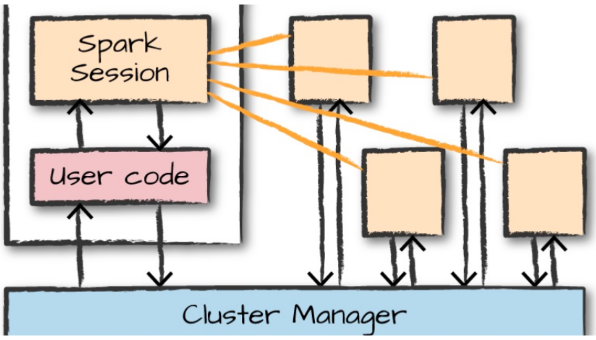
Dec 21, 2024 7 2

Diogo Santos

Mastering Apache Spark Shuffle Optimization: Techniques for...

Explore the power of bucketing, repartitioning, and broadcast joins to...

Jan 26 64



In Towards Dev by Prem Vishnoi(cloudvala)

Apache Spark Architecture :A Deep Dive into Big Data...

Agenda

Feb 6 97 1

MERGE INTO finally in PySpark

```
mergedDF = (
    spark.table("source_table")
    .mergeInto("target_table", "target.key = source.key")
    .whenNotMatched()
    .insertAll()
)
```

Hubert Dudek

MERGE in databricks—a few new improvements about which you...

Recently, we made a few improvements to the MERGE command, which you can not...

May 30 38 2

See more recommendations