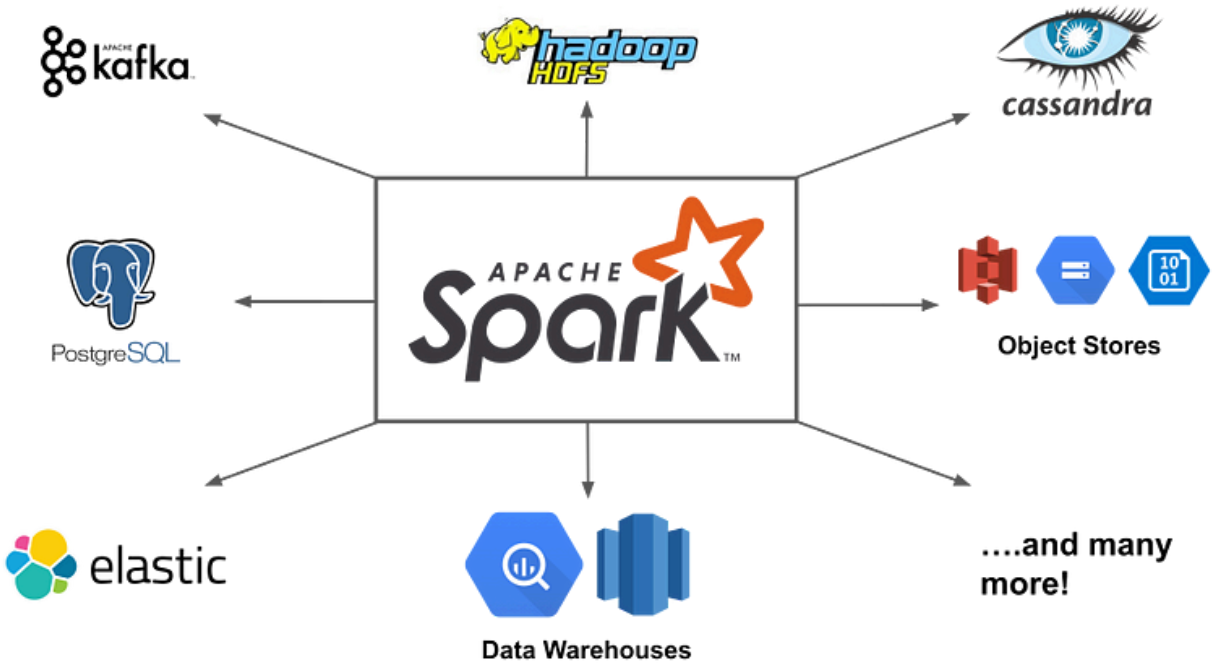
02 DAG in Spark - KirkYagami - Nikhil Sharma 👨‍💻🕵️‍♂️

DAG in Apache Spark



Apache Spark Ecosystem

Before understanding the theory about DAG, *let's talk about a real life scenario (day to day).*

What tasks do I need to complete in order to go to office?

- Wake up.
- Leave the bed.
- Get fresh.
- Take breakfast.
- Get ready.
- Go to office (Walk/Drive, etc.)

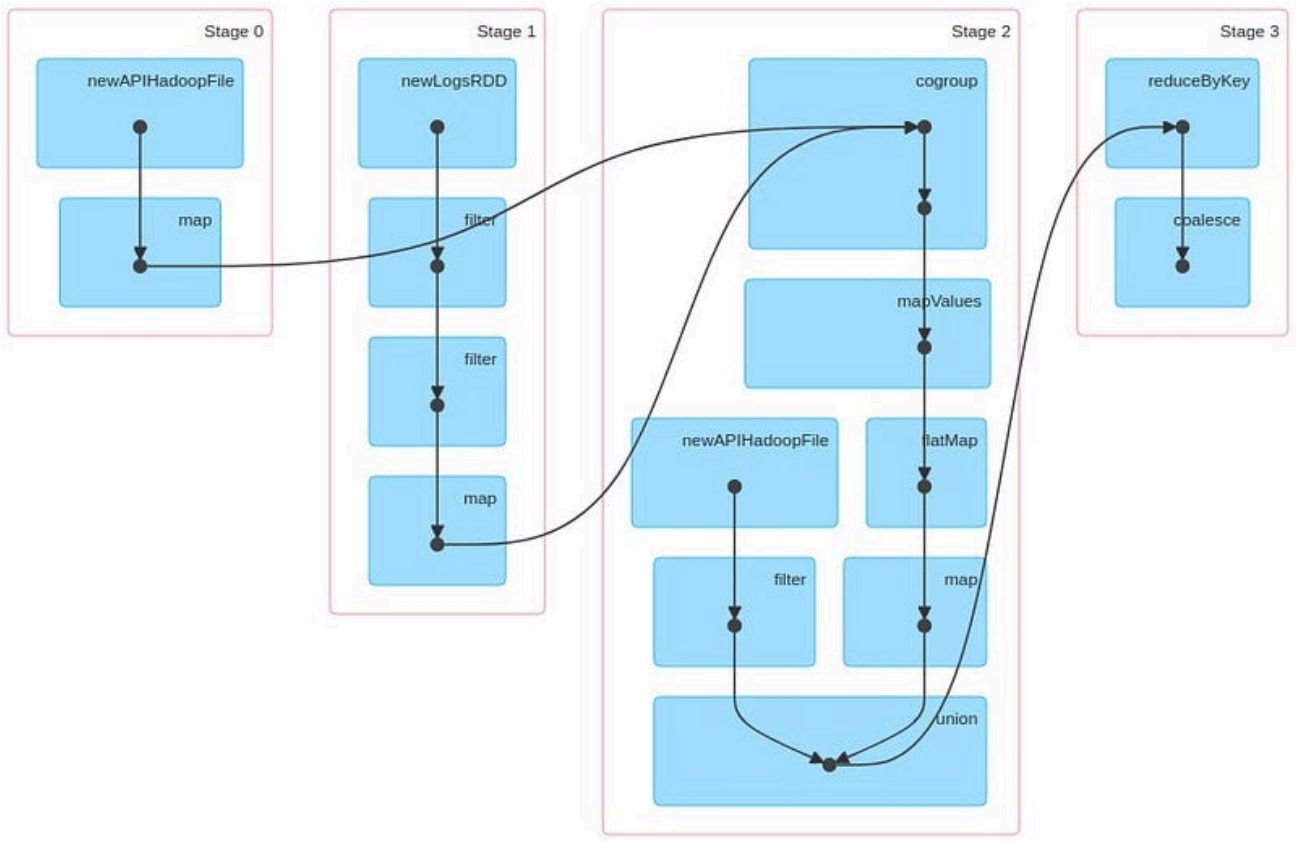
If I would need to represent these as a sequence of stages, here's how it will look:

> **Wake up → Leave the bed → Get fresh → Take breakfast → Get ready → Drive to office**

The above example can be thought of as a DAG (Directed Acyclic Graph) where:

- The sequence of events can be related to different **Stages** of an action which is, "Going to office".
- Each stage depends on the completion of the previous stage, **in ideal case**. (Well, nothing stops you to wake up and go to office directly. Get a breakfast there, then get fresh and then get ready in the office's changing room, LOL !!!).
- There is **no cycle** in this uni-directional sequence of stages.

DAG in Apache Spark



A Sample DAG visual taken from Spark UI

In Apache Spark, DAG stands for **Directed Acyclic Graph**. It is a fundamental concept used by Spark's execution engine to ***represent and optimize*** the flow of operations in a data processing job.

To understand DAG in Apache Spark, let's consider a real-life example of analyzing customer data in an e-commerce company.

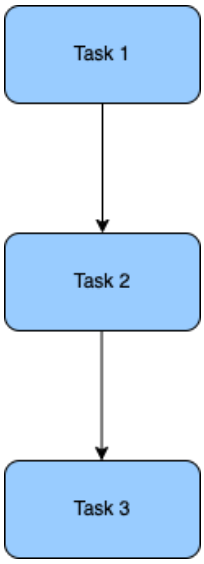
Suppose we have a dataset containing information about customer orders, such as order IDs, customer IDs, order dates, and the total amount spent.

Let's say we want to perform the following tasks on this dataset:

1. Task 1: Filter out orders that were placed in the last 30 days.

2. Task 2: Group the filtered orders by customer ID.
3. Task 3: Calculate the total amount spent by each customer.

These tasks can be represented as a DAG. ***Each task is represented as a node in the DAG, and the dependencies between tasks are represented as directed edges.***



Directed Acyclic Graph (No Cycle)

In this example, **Task 1** filters the orders based on the order date and produces a filtered dataset. **Task 2** takes the output of Task 1 and groups the orders by customer ID. Finally, **Task 3** takes the output of Task 2 and calculates the total amount spent by each customer.

The DAG is ***acyclic***, meaning there are ***no cycles or loops in the graph***. This property allows Spark to optimize and schedule the execution of the operations effectively, as it can determine the dependencies and execute the stages in the most efficient order.

Stages in DAG

Stages are a key concept within the Directed Acyclic Graph (DAG) execution model. A stage represents a ***set of tasks that can be executed together*** in a single wave of computation, resulting in a more efficient execution of the Spark job.

DAG Scheduler

It is the high-level scheduling layer that implements ***stage-oriented scheduling***.

- It computes a DAG of stages for each job, keeps track of which RDDs and stage outputs are materialized, and finds a minimal schedule to run the job.
- It then submits stages as *TaskSets* to an underlying *TaskScheduler* implementation that runs them on the cluster.

- It converts a logical execution plan (which consists of the RDD lineage formed through RDD transformations) into a physical execution plan.

Fault Tolerance through Spark DAG

If a worker node fails during the execution of a task, Spark's DAG scheduler can reschedule the failed task on another available node.

It uses the *lineage information* to identify the lost data and the pending tasks. By rescheduling the failed tasks on different nodes, Spark ensures that the computation continues from the point of failure, reducing the impact of failures on the overall job execution.

Optimization through Spark's DAG Optimizer

- **Task Fusion:** It can identify consecutive operations or transformations that can be combined into a single task, reducing the overhead of data shuffling and improving performance. This minimizes the data movement between stages and reducing the overall execution time.
- **Pipelining:** It can analyze the dependencies between tasks and identify opportunities for pipelining. It involves executing subsequent tasks as soon as their input data becomes available, without waiting for the completion of previous tasks. The latency between stages is improved and data processing throughput becomes better.
- **Stage Concurrency:** It can determine the independent stages within a job that can be executed concurrently. Stages that do not have any data dependencies can be executed in parallel, leveraging the available resources in the cluster efficiently.
- **Data Locality:** It takes data locality into account when scheduling tasks. It attempts to schedule tasks on the nodes where the data is already present or is being computed, minimizing data movement across the network. It reduces network overhead and improves performance.
- **Shuffle Optimization:** Spark's DAG optimizer applies various techniques to optimize data shuffling, which is often a costly operation in distributed data processing.

Conclusion

After going throught the details, we can easily conclude that the use of DAG in Apache Spark enables ***optimized execution, fault tolerance, and efficient scheduling of tasks, resulting in faster data processing and improved performance***.

Happy Learning !!!